Leo

Release 4.9

Edward K. Ream

February 27, 2013

Contents

1 Front Matter **3**

1.1 Acknowledgements 3

1.2 Leo's MIT License 4

2 前言 **5**

2.1 綱要結構爲真實資料 6

2.2 Leo 綱要非一般綱要 6

2.3 結論與鼓勵 7

3 What People Are Saying About Leo **9**

3.1 Leo is revolutionary 9

3.2 Leo is a showcase Python application 10

3.3 Leo is fun, even addicting 10

3.4 Leo is a flexible, powerful IDE 10

3.5 Leo is a superb outliner 11

3.6 Leo is an excellent PIM 11

3.7 Leo is a superb documentation tool 12

3.8 Leo simplifies the understanding of complex systems 12

3.9 Leo is stable, well designed and well supported 13

3.10 Longer quotes... 13

3.10.1 Speed Ream's slashdot article 13

3.10.2 Joe Orr 14

3.10.3 Dan Winkler 15

3.10.4 Dan Winkler 2 16

3.10.5 Dan Winkler 3 16

4 FAQ **19**

4.1 Getting Leo 21

4.1.1 Where can I get official releases of Leo? 21

4.1.2 How do I use bzr to get the latest sources from Leo's launchpad site? 21

4.1.3 How can I get recent bzr snapshots of Leo? 21

4.2 Installing Leo 22

4.2.1 Leo's installer failed, what do I do? 22

4.2.2 Nothing (or almost nothing) happens when I start Leo. What should I do? 22

4.2.3 Running Python setup.py install from the leo directory doesn't work. Why not? 22

4.3 Learning to use Leo . . . 22
4.3.1 What's the best way to learn to use Leo? . . . 22
4.3.2 Why should I use clones? . . . 22
4.3.3 When is using a section better than using a method? . . . 23
4.3.4 When is deleting a node dangerous? . . . 23
4.3.5 Why doesn't Leo support cross-file clones? . . . 24
4.3.6 How does EKR (Leo's developer) use Leo? . . . 24
Overview . . . 24
Additional tips . . . 25
Writing documentation . . . 25
Administrative tips . . . 26
Tips for using bzr . . . 26
4.3.7 How does Leo handle clone conflicts? . . . 26
4.4 Leo in Shared environments . . . 27
4.4.1 How should I use Leo with bzr/git/hg/svn/cvs? . . . 27
4.4.2 How can I use Leo cooperatively without sentinels? . . . 27
4.4.3 What's the recommended way to upgrade Leo? . . . 27
4.5 Using external files . . . 27
4.5.1 How do I inhibit sentinels in external files? . . . 27
4.5.2 How do I prevent Leo from expanding sections? . . . 28
4.5.3 How can I create Javascript comments? . . . 28
4.5.4 How can I disable PHP comments? . . . 28
4.5.5 How can I use Leo with unsupported languages? . . . 29
4.5.6 How do I make external files start with a shebang line? . . . 29
4.5.7 Can @file trees contain material not in the external file? . . . 30
4.5.8 How can I use Leo with older C compilers . . . 30
4.5.9 Why can't I use @ignore directives in @file trees? . . . 30
4.5.10 How can I avoid getting long lines in external files? . . . 31
4.6 Customizing Leo . . . 31
4.6.1 How can I add support for a new language? . . . 31
4.6.2 How do I submit a plugin? . . . 31
4.6.3 How do I add a new menu item from a plugin? . . . 31
4.6.4 How can I use Leo's legacy key bindings? . . . 32
4.6.5 How can I enable and disable support for psyco? . . . 32
4.6.6 How do I specify qt fonts? . . . 32
4.6.7 How do I set selection colors? . . . 32
4.7 Tips and techniques . . . 33
4.7.1 What is an easy way to profile code? . . . 33
4.7.2 How can I do a simple find and replace? . . . 34
4.7.3 How can I use Leo to develop Leo itself? . . . 34
4.7.4 How can I import many files at once? . . . 35
4.7.5 How can I use two copies of Leo to advantage? . . . 35
4.7.6 How can I display graphics in Leo? . . . 35
4.7.7 How can I create a template .leo file? . . . 36
4.7.8 How can I show Leo files with Excel? . . . 36

4.7.9 How can I reuse @button nodes in multiple files? . . . 37
4.7.10 How can I restore focus without using the mouse . . . 37
4.7.11 How can I make commonly-used scripts widely accessible? . . . 38
4.7.12 How can I use BibTeX citations from Leo? . . . 38
4.8 Trouble shooting . . . 38
4.8.1 How do I get help? . . . 38
4.8.2 How do I report bugs? . . . 38
4.8.3 My old .leo files won't load using Leo 4.5 or later. What should I do? . . . 39
4.8.4 Error messages from the rst3 plugin aren't helpful. What can I do? . . . 39
4.8.5 How can I run Leo from a console window? . . . 40
4.8.6 How can I use Python's pdb debugger with Leo? . . . 40
4.8.7 I can't write Imported files. What's going on? . . . 40
4.8.8 Nothing (or almost nothing) happens when I start Leo. What should I do? . . . 40
4.8.9 The new Python decorator syntax causes problems. What can I do? . . . 41
4.8.10 Running Python setup.py install from the leo directory doesn't work. Why not? . . . 41
4.8.11 I can't run the LeoBridge module outside of leo/core. What should I do? . . . 41
4.8.12 Why didn't Leo update my @shadow outline as expected? . . . 42
4.8.13 Why do Qt windows disappear in my scripts? . . . 42
4.9 Unicode issues . . . 42
4.9.1 I can not enter non-ascii characters. What can I do? . . . 42
4.9.2 Some characters in external files look funny. What can I do? . . . 42
4.9.3 I get weird results when defining unicode strings in scripts. What is going on? . . . 43
4.9.4 Some characters are garbled when importing files. What can I do? . . . 43
4.9.5 Python's print statement shows `byte hash' for unicode characters. What can I do? . . . 43

5 Slides **45**
5.1 Basic slide shows . . . 45
5.2 Intermediate slide shows . . . 45

6 Installing Leo **47**
6.1 Required and optional packages . . . 47
6.2 Installing other packages . . . 48
6.2.1 Installing Python . . . 48
6.2.2 Installing Qt . . . 48
6.2.3 Installing PyEnchant . . . 48
6.3 Installing Leo itself . . . 48
6.3.1 Installing Leo on Windows . . . 49
6.3.2 Installing Leo on Linux . . . 49
6.3.3 Installing Leo on MacOs 10.7 Lion . . . 49
6.3.4 Installing Leo with bzr . . . 50
6.4 Running Leo . . . 51
6.4.1 Running Leo the first time . . . 52
6.4.2 Running Leo in batch mode . . . 52
6.4.3 Running Leo from a console window . . . 52
6.4.4 The .leo directory . . . 52

7 The Leo Tutorial **55**
7.1 Leo's main window . . . 56
7.2 External files and @file nodes . . . 58
7.3 Creating external files from outlines . . . 59
7.3.1 Section references . . . 59
7.3.2 The @others directive . . . 60
7.3.3 The @all directive . . . 60
7.3.4 Choosing between @others and sections . . . 60
7.3.5 Organizing programs as outlines . . . 61
7.4 Clones & views . . . 62
7.5 More about directives . . . 62
7.6 Scripting, extending and customizing Leo . . . 64
7.7 Summary . . . 65
7.8 Further study . . . 65

8 Using Leo **67**
8.1 The minibuffer and minibuffer commands . . . 69
8.1.1 Basic editing commands . . . 70
8.1.2 Debugging commands . . . 71
8.1.3 Emacs commands . . . 71
8.1.4 Find commands . . . 74
8.1.5 Gui commands . . . 75
8.1.6 Help commands . . . 77
8.1.7 Mode commands . . . 77
8.1.8 Outline commands . . . 78
8.1.9 Miscellaneous commands . . . 80
8.2 Autocompletion . . . 81
8.2.1 Starting autocompletion . . . 82
8.2.2 Using the QCompleter . . . 82
8.2.3 Using the Log pane completer . . . 82
8.2.4 Showing docstrings . . . 83
8.2.5 Autocompleter settings . . . 84
8.2.6 Creating ctags data . . . 84
8.3 Calltips . . . 85
8.4 File commands . . . 85
8.4.1 Loading, Saving and Reverting Files . . . 85
8.4.2 Communicating with external editors . . . 86
8.4.3 Importing Files into Leo Outlines . . . 86
8.4.4 Exporting Files from Leo Outlines . . . 86
8.4.5 Quitting Leo . . . 86
8.5 Edit commands . . . 86
8.5.1 Undoing changes . . . 86
8.5.2 Cutting, pasting and selecting text . . . 87
8.5.3 Indenting body text . . . 87
8.5.4 Adding and deleting comments in body text . . . 87

8.5.5 Creating nodes from body text . . . 87
8.5.6 Converting leading blanks and tabs in body text . . . 87
8.5.7 Executing Python scripts in body text . . . 88
8.5.8 Finding and changing text . . . 88
Overview . . . 88
Basic searches . . . 88
find-all, clone-find-all & clone-find-all-flattened . . . 88
Change commands . . . 89
Incremental find commands . . . 89
Commands that set find options . . . 89
Word search and regex search commands . . . 89
Find settings . . . 90
8.5.9 Go To Line Number . . . 90
8.5.10 Inserting the date and time . . . 91
8.5.11 Reformatting paragraphs in body text . . . 91
8.5.12 Matching brackets and parenthesis . . . 91
8.5.13 Indenting body text automatically . . . 91
8.5.14 Creating and destroying multiple body editors . . . 92
8.5.15 Opening URL's . . . 92
8.5.16 Using chapters . . . 92
8.6 Outline commands . . . 93
8.6.1 Creating and destroying nodes . . . 93
8.6.2 Expanding & contracting nodes . . . 93
8.6.3 Cutting, pasting and deleting nodes . . . 93
8.6.4 Navigating through the outline . . . 94
8.6.5 Moving & Reorganizing nodes . . . 94
8.6.6 Cloning nodes . . . 94
8.6.7 Marking nodes . . . 94
8.6.8 Dragging nodes . . . 94
8.6.9 Hoisting & De-hoisting nodes . . . 95
8.6.10 Checking outlines . . . 95
8.6.11 Resizing panes . . . 95
8.7 Window commands . . . 95
8.8 Help commands . . . 96

9 Customizing Leo **97**
9.1 Specifying settings . . . 97
9.1.1 Configuration directories . . . 98
9.1.2 Search order for settings files . . . 98
9.1.3 Safe rules for local settings . . . 99
9.1.4 Organizer nodes . . . 99
9.1.5 @ignore and @if nodes . . . 100
9.1.6 Simple settings nodes . . . 100
9.1.7 Complex settings nodes . . . 101
@button . . . 101

@commands . . . 101
@data . . . 101
@enabled-plugins . . . 101
@font . . . 102
@menuat . . . 102
@menus . . . 103
@mode . . . 103
@recentfiles . . . 104
@shortcuts . . . 104
9.2 Input modes . . . 104
9.3 Adding extensible attributes to nodes and .leo files . . . 105
9.4 Translating Leo's menus and messages . . . 106
9.5 Writing new importers . . . 106

10 Controlling Syntax Coloring **109**
10.1 Syntax coloring settings . . . 110
10.2 Files . . . 111
10.3 The colorizer's inner loop . . . 112
10.4 Format of colorizer control files . . . 112
10.4.1 Ruleset names . . . 113
10.4.2 x.properties . . . 113
10.4.3 Attribute dictionaries and x.attributesDictDict . . . 113
10.4.4 Keyword dictionaries and x.keywordsDictDict . . . 114
10.4.5 Rules, rules dictionaries and x.rulesDictDict . . . 115
10.4.6 x.importDict and imported versus delegated rulesets . . . 117
10.5 Rule methods . . . 118
10.5.1 Arguments to rule methods . . . 118
10.5.2 match_eol_span . . . 118
10.5.3 match_eol_span_regexp . . . 119
10.5.4 match_keywords . . . 119
10.5.5 match_mark_following . . . 119
10.5.6 match_mark_previous . . . 120
10.5.7 match_seq . . . 120
10.5.8 match_seq_regexp . . . 120
10.5.9 match_span . . . 121
10.5.10 match_span_regexp . . . 121
10.5.11 match_terminate . . . 121

11 Creating Documents with Leo **123**
11.1 Tutorial . . . 124
11.1.1 Step 1: Create the @rst node . . . 125
Set global options . . . 125
Set your document's title . . . 126
11.1.2 Step 2: Write your document . . . 126
11.1.3 Step 3: Run the rst3 command . . . 127

11.1.4 Go forth and experiment . . . 127
11.2 Options . . . 127
11.2.1 General options . . . 127
11.2.2 Headline commands . . . 129
11.2.3 Option doc parts . . . 130
11.2.4 Defaults for options . . . 130
11.2.5 Http plugin options . . . 130
11.2.6 Section expansion options . . . 131
11.3 Other topics . . . 132
11.3.1 Markup doc parts . . . 132
11.3.2 Required cascading style sheets . . . 132
11.4 Further study . . . 132
11.5 Acknowledgements . . . 132
11.6 Theory of operation . . . 133

12 Scripting Leo with Python **135**
12.1 Basics . . . 137
12.1.1 c, g and p . . . 137
12.1.2 Import objects . . . 137
12.1.3 g.es writes to the log pane . . . 137
12.1.4 p.h and p.b . . . 138
12.1.5 c.redraw . . . 138
12.1.6 p.copy . . . 139
12.1.7 Generators . . . 139
c.all_positions & c.all_unique_positions . . . 140
p.children . . . 140
p.parents & p.self_and_parents . . . 140
p.siblings & p.following_siblings . . . 140
p.subtree & p.self_and_subtree . . . 141
12.1.8 Testing whether a position is valid . . . 141
12.1.9 g.pdb . . . 141
12.1.10 @button scripts . . . 142
12.1.11 autocompletion . . . 142
12.1.12 Summary . . . 143
Iterators . . . 143
Getters . . . 144
Setters . . . 144
12.2 Event handlers . . . 145
12.2.1 Enabling idle time event handlers . . . 149
12.3 Other topics . . . 149
12.3.1 g.app.windowList: the list of all open frames . . . 149
12.3.2 Ensuring that positions are valid . . . 149
12.3.3 g.openWithFileName . . . 150
12.3.4 g.getScript . . . 150
12.3.5 c.frame.body.bodyCtrl . . . 150

12.3.6 Invoking commands from scripts . . . 150
12.3.7 Getting settings from @settings trees . . . 151
12.3.8 Preferences ivars . . . 151
12.3.9 Functions defined in leoGlobals.py . . . 152
12.3.10 Making operations undoable . . . 152
12.3.11 Redirecting output from scripts . . . 152
12.3.12 Creating Qt Windows from Leo scripts . . . 153
12.3.13 Writing to different log tabs . . . 154
12.3.14 Invoking dialogs using the g.app.gui class . . . 154
12.3.15 Inserting and deleting icons . . . 154
12.3.16 Working with directives and paths . . . 155
12.3.17 Running Leo in batch mode . . . 155
12.3.18 Getting interactive input from scripts . . . 156
12.3.19 The @g.command decorator . . . 157
12.3.20 Modifying plugins with @script scripts . . . 158
12.3.21 Creating minimal outlines . . . 158
12.3.22 Retaining pointers to Qt windows . . . 159

13 Plugins **161**

13.1 Enabling plugins . . . 163
13.2 Summary . . . 163
13.3 Gui-independent plugins . . . 166
13.3.1 Commands & directives . . . 166
add_directives.py . . . 166
bzr_qcommands.py . . . 166
empty_leo_file.py . . . 166
import_cisco_config.py . . . 166
initinclass.py . . . 167
leo_interface.py . . . 167
lineNumbers.py . . . 167
macros.py . . . 167
mod_autosave.py . . . 168
mod_read_dir_outline.py . . . 168
mod_timestamp.py . . . 168
nodeActions.py . . . 169
outline_export.py . . . 171
paste_as_headlines.py . . . 171
pretty_print.py . . . 172
quickMove.py . . . 172
setHomeDirectory.py . . . 173
word_count.py . . . 173
13.3.2 Debugging . . . 173
debugger_pudb.py . . . 173
dump_globals.py . . . 173
enable_gc.py . . . 173

quit_leo.py . . . 173
trace_gc_plugin.py . . . 174
trace_keys.py . . . 174
trace_tags.py . . . 174
13.3.3 External programs . . . 174
ipython.py . . . 174
mod_tempfname.py . . . 174
open_shell.py . . . 174
tomboy_import.py . . . 175
vim.py . . . 175
xemacs.py . . . 176
word_export.py . . . 176
13.3.4 Files and nodes . . . 176
active_path.py . . . 176
at_folder.py . . . 177
at_produce.py . . . 177
at_view.py . . . 178
backlink.py . . . 178
datenodes.py . . . 178
expfolder.py . . . 178
FileActions.py . . . 179
geotag.py . . . 179
leocursor.py . . . 179
mime.py . . . 180
multifile.py . . . 180
niceNosent.py . . . 181
read_only_nodes.py . . . 181
run_nodes.py . . . 182
startfile.py . . . 182
xsltWithNodes.py . . . 182
13.3.5 Scripting . . . 183
dyna_menu . . . 183
leoscreen.py . . . 183
mod_scripting.py . . . 184
script_io_to_body.py . . . 186
13.3.6 Servers . . . 186
leoremote.py . . . 186
mod_http.py . . . 187
13.3.7 Slideshows and screenshots . . . 187
screenshots.py . . . 187
slideshow.py . . . 191
13.3.8 Text formatting . . . 191
bibtex.py . . . 191
dtest.py . . . 192
leo_to_html.py . . . 193

leo_to_rtf.py . . . 194
13.3.9 User interface . . . 194
UNL.py . . . 194
chapter_hoist.py . . . 195
detect_urls.py . . . 195
EditAttributes.py . . . 196
interact.py . . . 196
maximizeNewWindows.py . . . 196
mod_framesize.py . . . 196
plugins_menu.py . . . 196
redirect_to_log.py . . . 197
scripts_menu.py . . . 197
zenity_file_dialogs.py . . . 197
13.4 Qt only plugins . . . 198
13.4.1 attrib_edit.py . . . 198
13.4.2 colorize_headlines.py . . . 199
13.4.3 contextmenu.py . . . 199
13.4.4 nav_qt.py . . . 200
13.4.5 projectwizard.py . . . 200
13.4.6 quicksearch.py . . . 200
13.4.7 scrolledmessage.py . . . 200
13.4.8 spydershell.py . . . 201
13.4.9 stickynotes.py . . . 201
13.4.10 todo.py . . . 202
13.4.11 viewrendered.py . . . 202
13.4.12 graphcanvas.py . . . 203

14 Writing Plugins **205**
14.1 enabled-plugins . . . 205
14.2 Support for unit testing . . . 206
14.3 Important security warnings . . . 206

15 Unit testing with Leo **209**
15.1 Using @test nodes . . . 209
15.2 Using @suite nodes . . . 211
15.3 Using @mark-for-unit-tests . . . 211
15.4 How the unit test commands work . . . 212
15.5 @button timer . . . 212
15.6 @button profile . . . 212

16 Debugging with Leo **213**
16.1 Using g.trace and g.pdb . . . 213
16.2 Settings for winpdb . . . 214
16.3 Debugging scripts with winpdb . . . 214
16.3.1 The debug command . . . 214
16.3.2 The execute-script command with explicit debugger breaks . . . 214

17 Using @shadow **217**
17.1 Overview . . . 217
17.2 Creating @shadow trees . . . 218
17.3 What the update algorithm does . . . 218
17.4 Aha: boundary cases don't matter . . . 219

18 The leoInspect Module **221**
18.1 A query language . . . 222
18.2 Comparing leoInspect and Pylint . . . 223
18.3 Still to do . . . 223
18.4 Theory of operation . . . 223
18.4.1 Getters . . . 224
18.4.2 Computing token ranges . . . 224
Token-info prepass . . . 224
token_range . . . 225

19 Leo and Emacs **227**
19.1 Controlling Leo from Emacs using Pymacs . . . 227
19.2 Functions in leoPymacs.py . . . 229
19.3 The minibuffer . . . 230

20 IPython and Leo **231**
20.1 Introduction . . . 232
20.2 Installation and startup . . . 232
20.3 Accessing IPython from Leo . . . 233
20.3.1 IPython code . . . 233
20.3.2 Plain Python code . . . 234
20.4 Accessing Leo nodes from IPython . . . 234
20.4.1 LeoNode . . . 234
20.4.2 Accessing children with iteration and dict notation . . . 235
20.5 @cl definitions . . . 236
20.6 Special node types . . . 237
20.6.1 @ipy-startup . . . 237
20.6.2 @ipy-results . . . 237
20.6.3 @ipy-root . . . 237
20.6.4 @a nodes . . . 237
20.7 Launching ILeo from IPython . . . 238
20.8 Declaring custom push-to-ipython handlers . . . 238
20.9 Example code snippets . . . 239
20.10 Example use case: pylab . . . 239
20.10.1 Startup . . . 239
20.10.2 Usage . . . 240
20.11 Magic functions . . . 240
20.11.1 %mb . . . 240
20.11.2 %lee . . . 240
20.12 Acknowledgements and history . . . 241

21 Embedding Leo with the leoBridge module **243**
21.1 The basics 243
21.2 Running leoBridge from within Leo 245

22 Using Vim Bindings with Leo **247**
22.1 Installing vim bindings 247
22.2 General commands 247
22.3 Body pane commands 248
22.4 Outline commands 252
22.5 Commands not supported 255
22.6 Avoiding changes to `tag' files 256

23 Using ZODB with Leo **257**
23.1 Configuring Leo to use zodb 257
23.2 Initing zodb 258
23.3 Writing data to zodb 258
23.4 Defining zodb keys 258
23.5 Reading data from zodb 259
23.6 About connections 259
23.7 Convenience routines 260
23.7.1 g.init_zodb (pathToZodbStorage,verbose=True) 260
23.7.2 v.detach() 260

24 Leo's Reference **261**
24.1 Part 1: @<file> directives 261
24.1.1 @asis <path> 262
24.1.2 @auto <path> 262
24.1.3 @edit <path> 263
24.1.4 @file <path> (aka @thin) 264
24.1.5 @nosent <path> 264
24.1.6 @shadow <path> 264
24.2 Part 2: @all and @others 264
24.3 Part 3: Syntax coloring directives 265
24.4 Part 4: Dangerous directives 265
24.5 Part 5: All other directives 266

25 Designing with Leo **271**
25.1 Outlines embody design 271
25.2 Nodes hide details 272
25.3 Clones create views 272
25.4 Nodes create context 273

26 History of Leo **275**
26.1 Beginnings 276
26.2 Breakthrough 276
26.3 Apple and YellowBox 276
26.4 Borland C++ 276

26.5 Discovering Python . . . 277
26.6 SourceForge . . . 277
26.7 Allowing sentinel lines in external files . . . 277
26.8 Untangling @file is easy! . . . 277
26.9 Leo 3.x: Continuous improvement . . . 277
26.10 Leo 4.0: Eliminating error `recovery' . . . 278
26.11 Leo 4.1: The debut of gnx's . . . 278
26.12 Leo 4.2: Complete at last . . . 278
26.13 Leo 4.3 Settings . . . 279
26.14 Leo 4.4 The minibuffer and key bindings . . . 279
26.15 Leo 4.4.x Improvements . . . 279
26.16 Leo 4.5 @shadow files . . . 280
26.17 Leo 4.6 Caching, Qt and more . . . 280
26.18 Leo 4.7 The one node world and Python 3k . . . 280
26.19 Leo 4.8 Simple sentinels & better data recovery . . . 280

27 Theory of Operation 281
27.1 Autocompletion . . . 281
27.1.1 UI notes . . . 281
27.1.2 Appearance . . . 282
27.1.3 Performance . . . 282
27.2 Clones . . . 282
27.3 Drawing and events . . . 283
27.4 Find and change commands . . . 284
27.5 Key handling . . . 284
27.5.1 Key domains . . . 284
Domain 1: Parsing user bindings in @keys, @mode and @shortcut nodes . . . 285
Domain 2: Maintaining and using binding tables . . . 285
Domain 3: Translating incoming key events into standard events . . . 285
Domain 4: Printing key bindings in a human-readable format . . . 285
27.5.2 Key bindings . . . 285
27.5.3 Handling key events . . . 286
All events are key events . . . 286
c.check_event . . . 286
27.5.4 About the KeyStroke class . . . 287
27.5.5 Simplifying the Qt input code . . . 287
27.5.6 Typed dicts . . . 288
27.6 Nodes . . . 288
27.7 Overview . . . 288
27.8 Unicode . . . 289
27.9 Unlimited undo . . . 289
27.10 Leo's load process . . . 290

28 White Papers 291
28.1 Why I like Python . . . 291

28.1.1 Clarity . . . 291
28.1.2 Power . . . 292
28.1.3 Safety . . . 292
28.1.4 Speed . . . 292
28.1.5 Conclusions . . . 293

29 Appendices **295**
29.1 Format of .leo files . . . 295
29.2 Format of external files . . . 297
29.3 Unicode reference . . . 299

30 Glossary **301**

31 What's New in Leo **307**
31.1 Leo 4.10 . . . 308
31.1.1 New features & commands . . . 308
Weightless unit testing . . . 309
31.1.2 Plugins . . . 310
31.1.3 Scripts . . . 310
31.1.4 Settings . . . 310
31.1.5 Bugs fixed . . . 311
31.2 Previous versions . . . 312
31.2.1 Leo 4.9 . . . 312
Bugs fixed . . . 312
Deprecated/removed features . . . 312
Major improvements . . . 312
New and improved features . . . 316
New in 4.9 b2 . . . 318
New in 4.9 b3 . . . 318
New in 4.9 b4 . . . 321
New in 4.9 rc1 . . . 321
31.2.2 Leo 4.8 . . . 322
New sentinels . . . 322
Drag and drop files into Leo . . . 322
Scripting improvements . . . 322
Improved @url nodes . . . 322
New & improved commands . . . 323
New plugins . . . 323
New settings & command-line args . . . 323
Other improvements . . . 324
31.2.3 Leo 4.7 . . . 325
The one-node world . . . 325
Leo supports Python 3.x . . . 325
Improved file handling . . . 325
New command-line options . . . 325
New commands . . . 326

New settings . . . 326
Improved plugins . . . 326
31.2.4 Leo 4.6 . . . 326
Improved unit testing . . . 326
Improved file handling . . . 326
Improved handling of rST files . . . 327
New code features . . . 327
New command-line options . . . 327
New commands . . . 328
New and improved directives . . . 328
New settings . . . 328
Plugins . . . 329
31.2.5 Leo 4.5 . . . 329
Major new features . . . 329
Major code reorganizations . . . 329
Minor new features . . . 329
New settings . . . 330
31.2.6 Leo 4.4.8 . . . 330
New features . . . 330
New and improved plugins . . . 331
New settings . . . 331
31.2.7 Leo 4.4.6 . . . 331
New commands . . . 331
New features . . . 331
New settings . . . 331
31.2.8 Leo 4.4.5 . . . 331
Bug fixed . . . 331
New features . . . 332
New commands . . . 332
New settings . . . 332
31.2.9 Leo 4.4.4 . . . 332
The Great Graph Aha . . . 333
Added support for @auto files . . . 333
New commands for resolving cvs conflicts . . . 335
New kinds of settings trees . . . 335
New plugins . . . 336
Leo's core is now compatible with jython . . . 336
Improved prototype for icons in headlines . . . 336
Minor improvements . . . 336
Summary of new commands . . . 337
31.2.10 Leo 4.4.3 . . . 337
31.2.11 Leo 4.4.2 . . . 338
A major code reorg . . . 338
New commands . . . 338
New features . . . 338

New and improved plugins . . . 339
Settings . . . 340
ZODB scripting . . . 340
31.2.12 Leo 4.4.1 . . . 340
New commands . . . 341
New features . . . 341
New and improved plugins . . . 341
New settings . . . 342
Improved settings . . . 342
Minor improvements . . . 342
31.2.13 Leo 4.4 . . . 343
New commands . . . 343
New features . . . 345
New and improved plugins . . . 346
New and improved settings . . . 347

Leo's home page

Contents:

第 1 章

Front Matter

Contents

- Front Matter
  - Acknowledgements
  - Leo's MIT License

1.1 Acknowledgements

Leo owes much of its visual design to MORE, possibly the most elegant computer program ever written. Leo's clone nodes are inspired by MORE.

The following people have made generous donations to the Leo project: Robert Low, Nic Cave-Lynch.

The following people reported bugs, answered questions, and made suggestions for improving Leo: Alex Abacus, Shakeeb Alireze, Steve Allen, Bruce Arnold, Chris Barker, Dennis Benzinger, David Boddie, Jason Breti, Eric Brown, Terry Brown, Darius Clarke, Martin Clifford, Jason Cunliffe, Josef Dalcolmo, Gil Dev, Bill Drissel, Wenshan Du, Allen Edwards, Chris Elliot, Dethe Elza, Mark Engleberg, Roger Erens, Stephen Ferg, Tom Fetherston, Tomaz Ficko, Niklas Frykholm, Fred Gansevles, Jonathan M. Gilligan, Zak Greant, Thomas Guettler, Romain Guy, Dave Hein, Tiago Castro Henriques, Gary Herron, Steve Holden, Klass Holwerda, Matthias Huening, Robert Hustead, John Jacob, Paul Jaros, Christopher P. Jobling, Eric S. Johansson, Garold Johnson, James Kerwin, Nicola Larosa, David LeBlanc, Chris Liechti, Steve Litt, Martin v. Lwis (Loewis), Robert Low, Fredrik Lundh, Michael Manti, Alex Martelli, Marcus A. Martin, Gidion May, David McNab, Frank Merenda, Martin Montcrieffe, Will Munslow, Chad Netzer, Derick van Niekerk, Jeff Nowland, Naud Olivier, Joe Orr, Marc-Antoine Parent, Paul Paterson, Sean Shaleh Perry, Tim Peters, David Priest, Gary Poster, Scott Powell, Bruce Rafnel, Walter H. Rauser, Olivier Ravard, David Speed Ream, Rich Ries, Aharon Robbins, Guido van Rossum, David Rowe, Davide Salomoni, Steven Schaefer,Johannes Schn, Wolfram Schwenzer, Casey Wong Kam Shun, Gil Shwartz, Jim Sizelove, Paul Snively, Jurjen Stellingwerff, Phil Straus, David Szent-Gyrgyi, Kent Tenney, Jeffrey Thompson, Gabriel Valiente, Jim Vickroy, Tony Vignaux, Tom van Vleck, Kevin Walzer, Ying-Chao Wang, Cliff Wells, Dan Wharton, John Wiegley, Wim

Wijnders, Dan Winkler, Vadim Zeitlin.

The following have contributed plugins to Leo:

Rodrigo Benenson, Pierre Bidon, Felix Breuer, Terry Brown, Mike Crowe, Josef Dalcolmo, Michael Dawson, e, Roger Erens, Andrea Galimberti, Engelbert Gruber, Timo Honkasalo, Jaakko Kourula, Maxim Krikun, Zhang Le, LeoUser, Frric Momm, Bernhard Mulder, Mark Ng, Alexis Gendron Paquette, Paul Paterson, Dan Rahmel, Davide Salomoni, Ed Taekema, Kent Tenney, Brian Theado, Ville M. Vainio, Steve Zatz.

The following deserve special mention: David Brock wrote TSyntaxMemo. The late Bob Fitzwater kept me focused on design. Donald Knuth invented the CWEB language. Jonathan M. Gilligan showed how to put the Leo icon in Leo's windows. Joe Orr created XSLT stylesheets for Leo; see http://www.jserv.com/jk_orr/xml/leo.htm. Joe Orr also created an outstanding set of tutorials for Leo; see http://www.evisa.com/e/sb.htm. LeoUser (B.H.) contributed numerous plugins and was the inspiration for Leo's minibuffer. LeoUser also wrote jyLeo: Leo for Jython. The late Bernhard Mulder proposed a new way of untangling external files. John K. Ousterhout created tcl/Tk. Neal Norwitz wrote PyChecker. Marc-Antoine Parent urged me to use XML for Leo's file format and helped improve it. Paul Paterson suggested the plugin architecture, suggested an approach to spell checking and has contributed many excellent plugins. Frans Pinard wrote pymacs. Norman Ramsey created noweb and gave permission to quote from the noweb web documentation. Rich Ries has contributed a huge number of suggestions. Steven P. Schaefer pointed out major security problems lurking in hooks. Gil Shwartz helped with unicode support. Phil Straus has been a great friend and constant support. Guido van Rossum created Python, Tkinter and the Python License. Dave Winer created MORE. Ville M. Vainio created ILeo and has made many other valuable contributions to Leo. Dan Winkler helped support Leo on the Mac.

Special thanks to my family. My brother, David Speed Ream, tested Leo and made many useful suggestions. Rebecca, James and Linda make it all worthwhile. It was during a conversation with Rebecca that I realized that MORE could be used as a prototype for Leo. That was a crucial first step.

1.2 Leo's MIT License

All parts of Leo are distributed under the following copyright. This is intended to be the same as the MIT license, namely that Leo is absolutely free, even for commercial use, including resale. There is no GNU-like ``copyleft'' restriction. This license is compatible with the GPL.

第 2 章

前言

Contents

- 前言
  - 綱要結構爲真實資料
  - Leo 綱要非一般綱要
  - 結論與鼓勵

究竟爲甚麼有人會對 Leo 編輯器感興趣? 畢竟,Emacs 與 Vim 都是超級優的編輯器, 而且 Visual Studio 與 Eclipse 也都是很棒的 IDE's. Leo 憑甚麼與這麼強的對手競爭?Leo 又提供了哪些其他工具所沒有的功能?

Leo 確實具備一些特色 -- 是 Emacs, Vim, Visual Studio 與 Eclipse 所沒有的功能. 當 Leo 用戶了解這些特色之後, 就會直呼 Aha! 這句 Aha 來自將程式, 設計與資料, 以前所未有的方式進行處理. 姑且稱之爲"Leo 之道". 本質上,Leo 不只將程式, 設計與資料當作文字. 沒錯, 大家經常使用文字來表達程式, 設計與資料. 而大家也都以文字進行處理, 但是文字並不代表其中應該有的全部內涵.

以 Leo 之道來看, 文字僅是隱身其中, 屬於更重要內涵的一種表象 (影子). 意在言外的重點可以稱之爲" 組織" 或" 架構" 或" 觀點" 或甚至" 結構體". 這裡我們以 節點 這個詞來表示 Leo 結構中的基本單元. 接下來讓我加以說明.

以建築術語來說, 節點就如同建築物所使用的磚塊. 若在計算機程式術語來說, 節點組成方法, 類別, 檔案與整個應用程式. 因此所謂的" 節點" 並 * 不 * 是一個固定的意涵

- 而只是組織中的一個單元. 任一節點都能從其他節點來建構, 而且任一節點也都能

  爲其他節點所用.Leo 直接將節點表示爲大綱中的節點 (標題). 每一個大綱節點都包含標題 (headline) 與內文 (body text). **大綱視窗 ** 列出所有的標題; 而 ** 內文視窗 ** 則列出目前所選標題對應的內文.

2.1 綱要結構爲真實資料

再重申一次:Leo 中最基本單元並 * 不是 * 文字. 當然, 標題與內文都 * 包含 * 文字, 但是節點卻不是文字, 而是貨真價實的 (Python) 物件. 這代表幾個特點:

1. 由於節點爲真實的物件,Leo 指令知道節點的內容而且在整個大綱中的位置. 待會再來看大綱組織, 這裡先舉個例子. 每一個節點正好有一個父節點,(除了最頂層的節點外) 並且每一個節點都擁有零個或更多子節點, 以及零個或更多下屬節點. 一個節點的父, 子與下屬都是該節點的真實屬性, 並且完全獨立於該節點的標題或內文.

並且, 任一 Leo 指令 (或使用者所寫的程式與延伸套件) 都能 * 輕易 * 擷取某一標題的所有內涵, 而 * 無需 * 解讀節點內容. 指令, 程式碼與延伸套件都能輕易完成下列事項: 取得大綱的根節點, 大綱中所選擇的節點, 父節點, 下屬節點, 子節點或上屬節點等. 同時也能輕易插入, 刪除, 或移動節點, 也能變更任何節點的標題或內文. 所有這些動作都 * 無需 * 解讀節點文字內容.

2. 將節點視爲真實物件代表指令與延伸套件都能將標題文字視爲與內文 * 完全不同 * 的屬性. 此一特性將標題文字解讀爲內文的簡要說明. 這點非常重要! 標題經常用來控制 Leo 的指令. 例如, 標題以 @file, @asis, @auto 等開頭, 就可以用來引導 Leo 控制讀寫指令. 以 @test, @suite 與 @mark-for-unit-tests 開頭的標題, 則可用來執行 Leo 的單元測試. 並且, 也能輕易讓使用者建立新的標題規範, 用來控制使用者所寫的程式碼或延伸套件. 例如, 延伸套件定義以 @url, @rst, @bookmark, @slideshow 等開頭的標題功能. 因此將標題與內文分開而且 ** 當作真實節點物件 ** 的設計, 其實才是重點.

3. 另外值得一提的還有 Leo 程式延伸所提供的 @button 按鈕設計. 若標題爲 @button << 指令名稱 >>, 其內文帶有一組程式碼. 當 Leo 開啓一個 Leo 標題, 每一個 @button 節點可建立一個指令與圖像. 按下圖像 (或執行該指令) 就會讓所選擇的標題執行這段程式. 換言之, 程式的執行可以隨心所欲套用在所選擇的大綱某一部分. 讓使用者可以" 套用程式到某一資料", 亦即大綱內容的任何一部分. 尤其, 這也讓建立 ** 編輯程式 ** 來自動套用變得更加簡易, 而無需重複執行繁瑣無趣的程式編輯任務.

2.2 Leo 綱要非一般綱要

Earlier I said that *any* node can be built from other nodes, and *any* node can be used by any other node. It takes a very special kind of outline for this to be possible. In a typical outline, such as Emacs outline mode, for example, nodes appear exactly once in the outline. This makes it impossible to `reuse' nodes in multiple places. Leo removes that limitation: any outline node can be **cloned**, and clones can appear in as many places in an outline as you like. Although clones may look distinct on the screen, at the data level **each clone is exactly the same node**.

Earlier I said that you can think of nodes as representing `organization' or `structure' or `views' or even `architecture'. Clones are the crucial feature that allows this point of view. For example, we can build up multiple `views' of data in an outline using clones as follows:

- Create a `view node' that will represent a *user-specified* view.
- Clone all nodes that are to be part of the view, and move them so that each clone is a child of the view node.

That's about all there is to it. The view node, and its children *is* a new view of the outline. This notion of `view' is so important that Leo supports it directly. Leo's **chapters** are simply views created as I have just described. When you select one chapter, you only see the nodes of that chapter in Leo's outline pane.

2.3 結論與鼓勵

So Leo offers a new way to understand, organize and manipulate *any* kind of complex data, including computer programs, *designs* of computer programs, web sites, personal data, whatever. The Aha that I invite you to experience is this: Outlines are more than mere eye candy. Having organization be real data creates an entirely new dimension, literally and figuratively, in computer programming, computer design and data organization, including web-site design, database design, etc. Leo's commands use headline and body text in many creative ways. So can you and your scripts. It's easy, it's fun, and it's revolutionary.

That's about it, except for some words of caution and advice:

1. Leo has been under active development for over 10 years. The new world created by nodes is rich and varied. You won't learn it all in a day or so. Please be patient. Start by learning Leo's basic features as explained in the tutorial. You can learn more advanced features later.
2. Those of you who are comfortable with Emacs should feel pretty much at home with Leo. Leo has shamelessly stolen the best features of Emacs, including the minibuffer and many Emacs-like commands.
3. For those of you who are *not* comfortable with Emacs, please understand that you do *not* need to understand all of Leo's commands in order to use Leo. Start by ignoring the minibuffer. Later, the minibuffer can become your friend, but you can get the Aha! without it.

Edward K. Ream July, 2007

第 3 章

What People Are Saying About Leo

Contents

- What People Are Saying About Leo
  - Leo is revolutionary
  - Leo is a showcase Python application
  - Leo is fun, even addicting
  - Leo is a flexible, powerful IDE
  - Leo is a superb outliner
  - Leo is an excellent PIM
  - Leo is a superb documentation tool
  - Leo simplifies the understanding of complex systems
  - Leo is stable, well designed and well supported
  - Longer quotes...

3.1 Leo is revolutionary

``I am using Leo since a few weeks and I brim over with enthusiasm for it. I think it is the most amazing software since the invention of the spreadsheet."

``We who use Leo know that it is a breakthrough tool and a whole new way of writing code." -- Joe Orr

``I am a huge fan of Leo. I think it's quite possibly the most revolutionary programming tool I have ever used and it (along with the Python language) has utterly changed my view of programming (indeed of writing) forever." -- Shakeeb Alireza

``Thank you very much for Leo. I think my way of working with data will change forever... I am certain [Leo] will be a revolution. The revolution is as important as the change from sequential linear organization of a book into a web-like hyperlinked pages. The main concept that impress me is that the source listing isn't the main focus any more. You focus on the non-linear, hierarchical, collapsible outline of the source code." -- Korakot Chaovavanich

``Leo is a quantum leap for me in terms of how many projects I can manage and how much information I can find and organize and store in a useful way." -- Dan Winkler

``Wow, wow, and wow...I finally understand how to use clones and I realized that this is exactly how I want to organize my information. Multiple views on my data, fully interlinkable just like my thoughts." -- Anon

``Edward... you've come up with perhaps the most powerful new concept in code manipulation since VI and Emacs. -- David McNab

``Leo is...a revolutionary step in the right direction for programming." -- Brian Takita

3.2 Leo is a showcase Python application

``Thanks for a wonderful program – everybody should be using it! It blows the socks off that Java Mind mapping software that won project of the month a while back on sourceforge!" -- Derick van Niekerk.

``A few years back I would have said Zope was #1 Python showcase, but I agree 100% that Leo is tops now." -- Jason Cunliffe

``Leo is the most interesting Python project I know of...I see lots of stuff posted on the Daily Python page, but I usually yawn and come over to this forum to see what's cooking." -- Anon

``What an original synthesis of different ideas, why can't other Open Source projects change the way I think?" -- Anon

3.3 Leo is fun, even addicting

``When first I opened Leo, it was out of curiosity. But having used it...I'll never go back. They'll have to pry Leo out of my cold, dead fingers! Seriously, it should be renamed `Crack Cocaine' because it's that addictive. I'm ready to start a 12-Step group." -- Travers A. Hough

``I feel addicted to programming again...in fact [Leo] has resurrected a dead project of mine :) The Outline has proven most liberating in terms of testing ideas out." -- Anon

``I have been absolutely seduced by Leo over the past few days. I tell you, I can not put it down. I feel like a kid with a shiny new bike...I'm already bursting with new ways I'd like to use the tool in the future." -- Lyn Adams Headley

Thanks for the great work--I love Leo!!! -- Josef Dalcolmo

Leo has simplified updating and creating new scripts and .bats keeping similar information in the same place. there is almost an addictive withdrawal effect when I can complete an operation in so much less time with Leo & python than I had become used to. -- Anon

3.4 Leo is a flexible, powerful IDE

``[Leo] should either replace or greatly augment the development tools that I use." -- Zak Greant

``Leo is a marriage of outlining and programming. Pure genius. The main reason I am impressed with this tool is that it doesn't affect your choice of tools. You can use whatever IDE for whatever language and switch back and forth between Leo and it." -- Austin King

``Leo is the best IDE that I have had the pleasure to use. I have been using it now for about 2--3 months. It has totally changed not only the way that I program, but also the way that I store and organize all of the information that I need for the job that I do." -- Ian Mulvany

``I only have one week of Leo experience but I already know it will be my default IDE/project manager...people complain about the lack of a project manager for the free/standard Python IDE's like Idle. Leo clearly solves that problem and in a way that commercial tools can't touch." -- Marshall Parsons

``I have been using Leo for about 3 weeks and I hardly use my other programming editor anymore...I find it easy and enjoyable to use. I plan to adopt it as my presentation tool for code reviews." -- Jim Vickroy

``I'm absolutely astounded by the power of such a simple idea! It works great and I can immediately see the benefits of using Leo in place of the standard flat file editor." -- Tom Lee

I think you're really showing what open source can do and your current trajectory puts you on track to kick Emacs into the dustbin of computing history. -- Dan Winkler

3.5 Leo is a superb outliner

``Word outlines are very useful. But Leo makes Word look like a clunky toy." --Joe Orr

``Leo is an interactive editor for organizing text fragments hierarchically and sequentially into one or more files and hierarchical folders, without arbitrary limits on the number and size of text fragments and the depth of the hierarchy...Leo is a tool for combining hierarchically and sequentially organized text fragments into text files, hierarchically grouped into folders, with hierarchical or sequential organization of text within the files, and without arbitrary limits on the size and number of files and the depth of the hierarchy of folders and text nesting within the files." -- Alex Abacus

``Leo reminds me a great deal of things I loved when I used Userland's Frontier (an outlining cms with a native oodb) - but Frontier wasn't hackable enough for me, and it wasn't oriented towards coding..., and you couldn't round-trip rendered pages (big Leo win). This is really a super tool - in a matter of days I've started to use it on all my projects and I still haven't figured out how I lived without it." -- John Sequeira

``Leo is EXACTLY the kind of outliner I was looking for--fantastic job!" -- Steve Allen

``If you are like me, you have a kind of knowledge base with infos gathered over time. And you have projects, where you use some of those infos. Now, with conventional outliners you begin to double these infos, because you want to have the infos needed for the project with your project. With Leo you can do this too, but if you change text in one place IT IS UPDATED IN THE OTHER PLACE TOO! This is a feature I did not see with any other outliner (and I tried a few). Amazing! Leo directly supports the way I work!" -- F. Geiger

3.6 Leo is an excellent PIM

``Another day, another breakthrough using Leo--now I realize Leo is the best URL bookmark manager there is. No more bookmarks menus or favorites lists inside the browser for me. With the @url directive I can just double click on the URL to open it in my browser. Leo lets me arrange the URLs in a hierarchy (or multiple hierarchies), attach notes to them, save clippings of things I read on the sites. It's sooo much better than anything the browsers have built in and it lets me easily use different browsers on different platforms and different machines (try that with the browsers' built-in bookmark managers)." -- Dan Winkler

``I am an amateur photographer. I use plain old 35mm. film for my pictures. Over the weekend, I used Leo to organize my lists of pictures. It is quite helpful--I can have separate nodes for pictures I have enlarged, as well as pictures I have submitted to our local camera club. Thanks!" -- Rich Reis

``Cloning is pure genius!... Leo's cloning facility, allows me to create several views on the CFA course material. My main view follows the prescribed study guide. Another view is organized like the textbooks. Yet another gives me a glossary of terms. And when I'm done, I'll have some nice libraries...I can re-use later in other projects." -- Michael Manti

3.7 Leo is a superb documentation tool

``I've written documentation in WordPerfert, Ventura, Word, PageMaker, and FrameMaker and even though they create wonderfully looking and useful documents, they've never been able to do what I've been looking for. HTML, compiled help files, and later PDF came closer, but still not there...I think I've found it in LEO, a way to make a ``living" document. A document built out of discrete parts that can be re-organized on the fly to meet the needs of a varying audience...I've already started converting the IT Procedures manual from Open Office to LEO because I know it's going to be much more useful to me and anyone else...just the possibility of keeping system maintenance scripts in the IT manual is mind boggling." -- David Nichols

``With the help of the rst2 plugin, [Leo is] the best outliner I have yet encountered for writing the early stages of academic papers."

``A Leo file is an ideal documentation tool, collecting the assorted readme.txt files, the comments from the source files...as well as the config files themselves." -- Kent Tenney

3.8 Leo simplifies the understanding of complex systems

``Just as structured programming reveals and disciplines the flow control of a program, [Leo] allows the designer to reveal and discipline structure at many layers simultaneously: data structures, object structure, entity-relationship structure, client-server structure, design pattern structure, temporal structure, project management structure, and any other structure relevant to the system." -- Steven P. Schaefer

``A funny observation with Leo is that when I `Leo-ise' other people's code, Leo makes the code's structure so transparent that design faults become very quickly apparent. For example, maintenance pain caused by lack of factorization." -- David McNab

``Leo is a powerful tool for organizing text into tree structures, and for just generally attacking a number of problems from a tree-based perspective." -- Joe Orr

``I found this blog entry by someone (a talented former coworker of mine actually) complaining about some poorly written code she had to maintain: http://snippy.ceejbot.com/wiki/show/start/2003/01/29/001 She said: `You'd need a bulldozer to start refactoring it.' That was my cue to write a long message explaining that there is indeed such a bulldozer and it's called Leo. (You can see my message there as a reply to her original posting.) I gave her my recipe for how to get someone else's messy, scary code into Leo and how to break it down into manageable chunks." -- Dan Winkler

``Ed, you continue to push the envelope. The amazing thing is that the footprint isn't doubling every few months like it would be in another designer's hands. Adding features by removing constraints, hot refactoring while adding unit tests. Forget the book. I would pay to see the movie."

3.9 Leo is stable, well designed and well supported

``I am extremely impressed at how stable and useful Leo appears to be." -- Marcus A. Martin

``Leo is amazingly stable. Docs are often weak with Open Source Software. Not so Leo: Leo is unusually well documented." -- F. Geiger

``Leo is unimaginably useful and I always find new things it already knows(!) how to do. Indeed I am amazed by the never-ending resources and patience Edward is putting into it and its users community. Excellent." -- Gil Shwartz

I feel strongly that Ed Ream, our ever-patient, ever-productive Leo architect deserves a nomination [for the ActiveState OpenSource Award.] Among other reasons, for:

- Delivering the first usable visual literate programming tool.
- Adding a vast abundance of new features.
- Making possible a previously unimaginable amount of leverage in code editing.
- Eliminating vast amounts of menial programming labour.
- Tirelessly and patiently supporting users, and catering to a wide range of feature requests. -- David McNab

3.10 Longer quotes...

3.10.1 Speed Ream's slashdot article

September 3, 2002

Hello, my full name is David Speed Ream. I am known as Speed to friends and enemies alike, but I gladly answer to David or most any other handle. I am an unabashed and biased fan of Leo, the fact that it was written by my brother Edward only slightly coloring my already colored glasses. I have been testing and using Leo in software production for over 4 years. My company currently has over 50,000 lines of code in over 100 source files that are written using Leo.

My comments are from two points of view, the first being software project manager for a complicated, multi-module software product, and the second being as a production line coder. For me, Leo' s greatest and only real drawback is the learning curve. This learning curve can be shallow is if all that is required is that someone code using Leo. However, in our company we allocate 40 to 80 hours *on top* of the normal coding load for someone to come up to speed on Leo. The ROI (return on investment) is calculated by me to be on the order of 3 months. So if I hire a consultant for less than 3 months, I don' t teach him Leo, even though all source code in our company must reside in Leo files for the reasons I won' t go into now.

I consider that my coders are 15 to 30 percent more efficient in their daily operations than my competition' s people. This indefensible claim of mine is based on the changes in my productivity as Leo grew from a test document production tool to the primary production method for all our assembly, c and cpp source code.

Personally, I hate to deal with documentation when I write code, except:

1. When I am first sitting down to solve a new problem. Then the documentation becomes quite long-winded and pontificatory, as if I were the only one on earth smart enough to solve the problem - or
2. When I come back to code I or someone else has written and find the documentation insufficient to understand the code without study (seems to be most of the time).

So I do not require my engineers or myself to do a great job of documentation, nor do I use Leo for that purpose. Rather, it is Leo’ s outlining and organizing ability, and Leo’ s ability to create source files from within the outline that give me what I think is a tremendous competitive advantage. Each of my company’ s products run on all versions of windows from Win 3.1 to XP. In our flagship software piece, there are ten main modules, and each module is maintained by one single Leo file. In the CODEC module, one Leo file named compress.leo organizes and creates seven .asm files, forty-four .c files, twenty .h files, two .def files, four .mak files, etc. etc. etc. This one file can be checked out from source code control and given to an engineer for the addition of a new feature.

In it are contained all the known issues for the CODEC, each issue arranged in its own clone section. One clone section groups together every routine, variable or type definition that must change between different versions of Windows. These sections could be from six different c source files, two assembly files, and eight .h files. Another clone section groups together those sections relating to memory problems, which change according to the memory configuration and TSR configuration (or lack thereof) on the target machine. Another clone section groups sections that fail (or don’ t fail) if the routine in question was accidentally run during the dreaded ‘interrupt time’ . Another clone section is a section containing clones, each of which is named after the major bug that was fixed when the engineer who fixed the bug grouped a bunch of routines, definitions, etc. together to fix the bug.

None of the above clone sections was ‘designed’ into the document. Just the opposite happens. When the codec was first written, there was just a single Leo file with a bunch of sections for each c routine or assembly module. As the product grew and was tested on various platforms, each failure of the module was organized into clones each time a failure was fixed. This is what I call “SELF DOCUMENTING CODE” . This has nothing to do with me sitting and documenting anything. Its just that the STRUCTURE of a bug fix (or product enhancement) lives on long after the coding is done, as long as no one is foolish enough to delete the cloned sections that ‘DOCUMENT’ what happened.

In actual practice, this organizational ‘history’ is so powerful that I can’ t begin to describe it. A ‘REVERSE LEARNING CURVE’ happens when an engineer gets a Leo file that already has the ‘interrupt time sensitive’ routines grouped together by the last unfortunate soul who had to work on them. There may not be any more written documentation, but the knowledge contained in the structure can be breathtaking. It is certainly time saving. I find this particularly true in my own case. Often I’ ll look at some code that seems totally unfamiliar and think ‘what idiot wrote this crap’ . Then I’ ll look at the version control comments and realize that I wrote the crap. Then for sure I know the documentation is non-existent, but the clones I used to develop it are still there, and they always serve to refresh my memory in an indescribable way.

Enough of this commentary, I just looked at the clock. Best wishes to anyone willing to try Leo for a week. I hope you will be glad you did.

3.10.2 Joe Orr

The Word outlines are very useful. But Leo makes Word look like a clunky toy.

#1 Reason would probably be clone nodes. One node can point to another. Another way of putting this is is that a leaf can be on more than one tree. For example, suppose you have a list of recipes. You simultaneously put a single recipe under multiple categories or even multiple hierarchies. You could put ``3 bean enchilada'' simultaneously under Recipes-Mexican and Food-Gas. Another example would be, if you are a biologist trying to decide under which genus to put a new species, you could put the species under two simultaneously. In effect, you can build a 3-D tree. For a further illustration see http://www.3dtree.com/ev/e/sbooks/leo/sbframetoc_ie.htm

#2 Reason would probably be that Leo outlines can be embedded in external text files. So, a Leo outline is more than an outline, it is a meta-structure that can be added to another text without changing that text, but rather providing an external road map to the text. Microsoft Word has a text (xml) version with a commenting convention, so Leo can even be used to add outlines into Word docs, although it isn't set up to do that now. For example, see http://www.3dtree.com/ev/e/sbooks/leo/sbframetoc_ie.htm In this case, the upper window of Leo is the meta-structure, and the bottom window is the file to which the meta-structure is being applied, viewed one node at a time.

I may not have made #2 very clear, but it is actually a very useful feature. It takes some getting used to before one sees all of the possibilities tho. One way to think of it is that Leo allows you to throw external documents into your outline, and yet the external document remains independent and can still be edited separately.

Some other cool things about Leo which Word doesn't feature: 1. Pure xml output that is easy to transform into other formats (next version of Word will have true XML format, but not as easy to work with). One consequence of this is that Leo files can be transformed pretty easily to web pages with their outlining capability intact. 2. Easy to add features since is programmed in Python and open source. Maybe your average user can't start hacking on it, but a surprising amount can be tacked on... .. by flipping through the Tk manual. 3. Free, opensource, multi-platform 4. Leo is scriptable with Python. It should be possible to build a Tickler into Leo using Python scripting, for example.

3.10.3 Dan Winkler

First of all, kudos to you for the excellent progress you've been making with Leo. I upgraded today after about three months of using and older version and I was thrilled to see all the great improvements that have happened so fast. I especially love the ability to go to next clone. I think you're really showing what open source can do and your current trajectory puts you on track to kick Emacs into the dustbin of computing history.

So today I copied all my data (personal information manager and project management stuff) out of my old outliner (ThoughtManager, which syncs with and runs on the Palm) and put it into Leo. It took me hours to do it and then to rearrange it the way I really wanted it. But having the ability to make clones and have different ways to view my data is, as you know, fabulous. In my case, for personal information and project management things, I used the flexibility of clones to allow me to see my data in several different views: 1) by project, the logical hierarchical breakdown by topic, 2) by person, so whenever I'm talking to someone I can easily see all the pending items related to them which may be spread over multiple projects, 3) by priority, so I can see what needs to get done sooner and what can wait for later and, 4) a special case of priority called ``Today'' for the things I'm going to focus on in the coming hours.

Now here's why I don't miss the ability of my old outliner to synch the entire outline with the Palm. It turns out the main thing I really want in the Palm is the top category ``Today'' so all I have to do is have Leo flatten that one heading into a text file (and it kindly remembers the name and directory of the file I used last time) and then I'm done because I've told the Palm Hotsync manager that that file should be sent to Palm memo pad every time I synch. The Palm Hotsync manager does a nice job of sending a text file to the Palm memo pad and even breaks the file up into multiple memo records if it's too big to fit in just one. So that gives me enough to be able to browse (or full text search) the small amount of data that I really want right inside my Palm (which is also my cell phone). Quick and dirty but it works.

For times when I want my whole outline with me, Leo wins again because thanks to its cross platform nature I can take my whole outline with me on my Mac iBook, even though I usually edit it on a Windows PC (which is the only kind of machine my old outliner would run on). Quite frankly, although my old outliner was able to shoehorn the whole thing into my palm/cellphone, it was a pain to access it on the small screen and slow processor. Now when I anticipate I'll need the whole thing, for example when I'm going to a meeting, I can put it on my Mac iBook (under X and Fink for now until Python can do it native under Aqua) and have real, full access to it all.

I think now in addition to being great for programming Leo is also a great PIM. Being able to flatten a strategically chosen portion of the outline into a known file name that the Palm synch manager has been told to send to the Palm on every synch does the trick for me. I wonder if you would consider something like an @flatten directive so I can have that done automatically for me every time I save my outline? For now it's up to me to flatten the node I want manually, although once I've done that the transfer to the Palm is automatic.

You're my hero! Thank you so much.

3.10.4 Dan Winkler 2

Another day, another breakthrough using Leo -- now I realize Leo is the best URL bookmark manager there is. No more bookmarks menus or favorites lists inside the browser for me. With the @url directive I can just double click on the URL to open it in my browser. Leo lets me arrange the URLs in a hierarchy (or multiple hierarchies), attach notes to them, save clippings of things I read on the sites. It's sooo much better than anything the browsers have built in and it lets me easily use different browsers on different platforms and different machines (try that with the browsers' built-in bookmark managers).

When using Leo as a project manager and personal information manager as I do I can heavily annotate every task and project with helpful and relevant URLs. And since URLs can be of the file:// form, they're not just for web pages or HTML documents; I can link to any file on my disk of any type to be opened by any program.

Leo is a quantum leap for me in terms of how many projects I can manage and how much information I can find and organize and store in a useful way. I'm a data-mining army of one now and the web is my playground. Every time I find a web page that has interesting links to others, those links get stored in my Leo outline too, right where I can find them and make practical use of them. I can easily accept dozens of valuable links every day and integrate them into what I'm doing in a way that I'm confidant they won't get lost or forgotten. Before I always used to get bogged down by the difficulty of managing bookmarks inside the browser. But now I'm no longer the victim of information overload buried in the knowledge landslide of the Internet; instead I'm the professional strip miner with the world's biggest bulldozer. I eagerly plunge into mountains of data and emerge with all the valuable information nuggets neatly stored and organized. And my storehouse of knowledge is a flexible thing where I can reorganize and prioritize and massage the data to my heart's content as I learn more about it and decide to use it in different ways for different purposes. It's the difference between the pick axe and the steam shovel for me.

3.10.5 Dan Winkler 3

This year my accountant is getting a beautiful printout generated by LaTeX and Leo. I have a complicated tax situation this year, but I got it all laid out and organized in Leo. Then I had each of the nodes that had something my accountant needs to see write the data out to a file in the form a LaTeX table.

Sometimes a row of a table would have a result that was calculated by adding up a list of numbers. For that I used the modern day equivalent of an adding machine paper tape -- I stored a lisp s-expression in a Leo comment. I like s-expressions for this because once I put the opening ``(+" on one line and the closing ")" on another line, I can fill in additional numbers just by typing them and can even annotate them with comments. So in the middle of generating a LaTeX file I might have something like this:

```
@
(+
1165.26 1823.70 ; May 2002
123.38 ; June 2002
13.50 ; July 2002
13.21 ; October 2002
55.25 ; November 2002
)
@c
```

That's an annotated record of how I arrived at the number the accountant will actually see. I can just paste it into any lisp or scheme interpreter and get the total. Adding additional numbers is easy.

For next year, I think I might take this a step further. What I did this year is good for adding up numbers to get a total for one row of a LaTeX table. But it turns out I'd also like some more processing done on those tables (which I had to do by hand this time) -- I'd like the rows sorted in reverse order by magnitude (so that the big numbers jump out at you from the start of the tables) and I'd like a total of all the rows in the table. So I think next year, instead of having an s-expression that computes the total of one row for me, I think I'll use s-expressions that generate whole tables, formatted for LaTex, from the underlying data. So I'm thinking next year my s-expressions might look more like this:

```
@
(table "Widget Related Expenses"
   ("widget insurance" (+
                1165.26 1823.70 ; May 2002
                123.38 ; June 2002
                13.50 ; July 2002
                13.21 ; October 2002
                55.25 ; November 2002
               ))
    ("widget shipping" (+
                472.15 651.94 ; May 2002
                54 ; June 2002
               ))
    ("widget cleaning" (+
                165.26 183.70 ; May 2002
                123.38 ; June 2002
                13.50 ; July 2002
                13.21 ; October 2002
                55.25 ; November 2002
               ))
)
@c
```

The job of that ``table" function would be to return the LaTeX code needed to display a table with the category names and values, sorted descending by magnitude, with the total displayed. It's sort of a poor man's way of doing a spreadsheet inside Leo and then making it look great using LaTeX. The idea would be as I wanted to add more data, I'd add it to the s-expression and then reevaluate the whole thing by pasting it into a lisp interpreter and then copying the result back into the same Leo node for LaTeX to process.

-- Dan

第 4 章

FAQ

This is Leo's Frequently Asked Questions document.

Contents

- FAQ
  - Getting Leo
    * Where can I get official releases of Leo?
    * How do I use bzr to get the latest sources from Leo's launchpad site?
    * How can I get recent bzr snapshots of Leo?
  - Installing Leo
    * Leo's installer failed, what do I do?
    * Nothing (or almost nothing) happens when I start Leo. What should I do?
    * Running Python setup.py install from the leo directory doesn't work. Why not?
  - Learning to use Leo
    * What's the best way to learn to use Leo?
    * Why should I use clones?
    * When is using a section better than using a method?
    * When is deleting a node dangerous?
    * Why doesn't Leo support cross-file clones?
    * How does EKR (Leo's developer) use Leo?
    * How does Leo handle clone conflicts?
  - Leo in Shared environments
    * How should I use Leo with bzr/git/hg/svn/cvs?
    * How can I use Leo cooperatively without sentinels?
    * What's the recommended way to upgrade Leo?
  - Using external files
    * How do I inhibit sentinels in external files?
    * How do I prevent Leo from expanding sections?
    * How can I create Javascript comments?
    * How can I disable PHP comments?
    * How can I use Leo with unsupported languages?
    * How do I make external files start with a shebang line?
    * Can @file trees contain material not in the external file?
    * How can I use Leo with older C compilers
    * Why can't I use @ignore directives in @file trees?
    * How can I avoid getting long lines in external files?
  - Customizing Leo
    * How can I add support for a new language?
    * How do I submit a plugin?
    * How do I add a new menu item from a plugin?
    * How can I use Leo's legacy key bindings?
    * How can I enable and disable support for psyco?
    * How do I specify qt fonts?
    * How do I set selection colors?
  - Tips and techniques
    * What is an easy way to profile code?
    * How can I do a simple find and replace?
    * How can I use Leo to develop Leo itself?
    * How can I import many files at once?
    * How can I use two copies of Leo to advantage?
    * How can I display graphics in Leo?
    * How can I create a template .leo file?

4.1 Getting Leo

4.1.1 Where can I get official releases of Leo?

You can get the latest official releases of Leo at http:// sourceforge.net/ project/ showfiles.php? group_id=3458&package_id=29106

However, if at all possible, it is better to use bzr to get the latest sources. See the next entry.

4.1.2 How do I use bzr to get the latest sources from Leo's launchpad site?

Many users will want to track the development version of Leo, in order to stay on top of the latest features and bug fixes. Running the development version is quite safe and easy, and it's also a requirement if you want to contribute to Leo.

1. First, you need to get Bazaar (bzr) from http://bazaar-vcs.org. For windows users we recommend the standalone installer - the python installer may have problems pushing to Launchpad. Plain bzr installer only contains the command line version, so you might want to augment that with a friendly GUI - qbzr is recommended as it's the easiest one to install. It provides command like `bzr qlog', `bzr qannotate' etc.
2. Get Leo from launchpad by doing:

```
bzr branch lp:leo-editor
```

And that's it! You can run the launchLeo script (in the top-level branch directory) directly. When you want to refresh the code with latest modifications from Launchpad, `run bzr pull'.

If you make modifications to Leo (with the interest in sharing them with the Leo community), you can check them in to your local branch by doing `bzr checkin'. Now, to actually request your changes to be merged to Leo trunk, you need a Launchpad account with RSA keys in place. There is showmedo video about how to accomplish this on Windows using puttygen and pageant at http://showmedo.com/videos/video?name=1510070&fromSeriesID=151.

After your Launchpad account is set up, go to https://launchpad.net/leo-editor, choose Code tab -> Register Branch, select Branch type ``Hosted'' and fill in descriptive details about the branch. After that, go to the branch home page from Code tab again, and copy-paste the push command line to terminal. For example, for branch:

```
https://code.launchpad.net/~leo-editor-team/leo-editor/mod_rclick
```

The push command is:

```
bzr push bzr+ssh://my_name@bazaar.launchpad.net/~leo-editor-team/leo-editor/mod_rclick
```

You may wish to add --remember command line option to bzr push, to direct all future pushes to that location. Then, you only need to execute `bzr push'.

After your branch is pushed, you can email the Leo mailing list and request it to be reviewed and merged to trunk.

-- Ville M. Vainio - vivainio.googlepages.com

4.1.3 How can I get recent bzr snapshots of Leo?

Daily snapshots are available at http://www.greygreen.org/leo/

4.2 Installing Leo

4.2.1 Leo's installer failed, what do I do?

You can simply unpack Leo anywhere and run from there. You don't need the installer.

From a console window, cd to the top-level leo folder. Run Leo as follows:

```
python launchLeo.py
```

To run Leo with Qt look and feel, use the --gui=qt option:

```
python launchLeo.py --gui=qt
```

To load Leo's source, load leoPyRef.leo:

```
python launchLeo.py --gui=qt leo\\core\\leoPyRef.leo
```

4.2.2 Nothing (or almost nothing) happens when I start Leo. What should I do?

Missing modules can cause installation problems. If the installer doesn't work (or puts up a dialog containing no text), you may install Leo from the .zip file as described at How to install Leo on Windows. However you are installing Leo, be sure to run Leo in a console window. because as a last resort Leo prints error messages to the console.

4.2.3 Running Python setup.py install from the leo directory doesn't work. Why not?

Leo's setup.py script is intended only to create source distributions. It can't be used to install Leo because Leo is not a Python package.

4.3 Learning to use Leo

4.3.1 What's the best way to learn to use Leo?

First, read the tutorial. This will be enough to get you started if you just want to use Leo as an outliner. If you intend to use Leo for programming, read the Quick start for programmers, then look at Leo's source code in the file LeoPy.leo. Spend 5 or 10 minutes browsing through the outline. Don't worry about details; just look for the following common usage patterns:

- The (Projects) tree shows how to use clones to represent tasks.
- Study @file leoNodes.py. It shows how to define more than one class in single file.
- Most other files show how to use a single @others directive to define one class.
- Most methods are defined using @others, *not* section definition nodes.

4.3.2 Why should I use clones?

You will lose much of Leo's power if you don't use clones. See Clones & views for full details.

4.3.3 When is using a section better than using a method?

Use methods for any code that is used (called or referenced) more than once.

Sections are convenient in the following circumstances:

- When you want to refer to snippets of code the can not be turned into methods. For example, many plugins start with the code like this:

  ```
  << docstring >>
  << imports >>
  << version history >>
  << globals >>
  ```

  None of these sections could be replaced by methods.

- When you want to refer to a snippet of code that shares local variables with the enclosing code. This is surprisingly easy and safe to do, *provided* the section is used only in one place. Section names in such contexts can be clearer than method names. For example:

  ```
  << init ivars for writing >>
  ```

In short, I create sections when convenient, and convert them to functions or methods if they need to be used in several places.

4.3.4 When is deleting a node dangerous?

A **dangerous** delete is a deletion of a node so that all the data in the node is deleted *everywhere* in an outline. The data is gone, to be retrieved only via undo or via backups. It may not be obvious which deletes are dangerous in an outline containing clones. Happily, there is a very simple rule of thumb:

```
Deleting a non-cloned node is *always* dangerous.
Deleting a cloned node is *never* dangerous.
```

We could also consider a delete to be dangerous **if it results in a node being omitted from an external file.** This can happen as follows. Suppose we have the following outline (As usual, A' indicates that A is marked with a clone mark):

```
- @file spam.py
  - A'
    - B
- Projects
  - A'
    - B
```

Now suppose we clone B, and move the clone so the tree looks like this:

```
- @file spam.py
  - A'
    - B'
- Projects
  - A'
```

```
    - B'
  - B'
```

If (maybe much later), we eliminate B' as a child of A will get:

```
- @file spam.py
  - A'
- Projects
  - A'
  - B
```

B has not been destroyed, but B is gone from @file spam.py! So in this sense deleting a clone node can also be called dangerous.

4.3.5 Why doesn't Leo support cross-file clones?

Cross-file clones are cloned nodes in one outline that refer to data in another outline. This is a frequently requested feature. For example:

```
I would absolutely love to have the leo files in different project
directories, and a "master" leo file to rule them all.
```

However, cross-file clones will never be a part of Leo. Indeed, cross-file clones would violate the principle that data should be defined and managed in exactly one place. Just as human managers would not willingly accept shared responsibility for even a single line of code, every piece of Leonine data should be the responsibility of one and *only* one .leo file.

The problem fundamental. If the *same* (cloned) data were ``owned" by two different Leo files we would have a classic ``multiple update problem" for the data. Each outline could change the data in incompatible ways, and whichever outline changed the data last would ``win."

To make such a scheme workable and safe, one would have to devise a scheme that would keep the data in ``component" .leo files consistent even when the component .leo files changed ``randomly", without the ``master" .leo file being in *any* way in ``control" of the changes. Good luck :-)

Let us be clear: it's no good having a scheme that works *most* of the time, it must work *all* the time, even with unexpected or even pathological file updates. If it doesn't you are asking for, and will eventually get, catastrophic data loss, without being aware of the loss for an arbitrarily long period of time. Even with a source code control system this would be an intolerable situation.

4.3.6 How does EKR (Leo's developer) use Leo?

Here is the workflow I use to develop Leo. The intention is to help present and potential developers use Leo effectively.

Overview

- Develop in an outline containing all of Leo's source files. Close this outline rarely: this keeps the code I am using stable while I'm hacking the code.

- Test in a *separate* .leo file, say test.leo. In fact, I often test in a private file, ekr.leo, so that test.leo doesn't get continually updated on bzr with trivial changes.

These two points are covered in a bit more detail in This FAQ entry.

Additional tips

A. Avoid using the mouse whenever possible. For example, use alt-tab to switch between windows.

2. Always develop Leo in a console. This allows you to see the output of g.trace.

Speaking of g.trace, I hardly ever use `print' because g.trace prints the name of the function or method in which it appears. The typical pattern for enabling traces is:

```
trace = True and not g.unitTesting
if trace: g.trace(whatever)
```

This pattern is especially useful when a method contains multiple calls to g.trace.

3. I use scripts to open particular Leo files. These are batch files on Windows, and aliases on Linux, but invoking them is the same on either platform:

```
all:    opens all my main development files using the qt-tabs gui.
t:      opens test.leo.
e:      opens ekr.leo.  I use this file for private testing.
d:      opens LeoDocs.leo.
s:      opens LeoPy.leo.
plugins: opens leoPlugins.leo.
gui:    opens leoGui.leo.
u:      opens unitTest.leo.
```

   These run Leo with Python 3.x. There are similar scripts, ending in 2, that run Leo with Python 2.x. For example, u2 opens unitTest.leo with Python 2.x. Thus, to run a test, I alt-tab to an available console window, then type `e' or `t' or `u' or, if I want Python 2.x, `e2' or `t2' or `u2'.

4. Use clones to focus attention on the task at hand. For more details, see about clones and views.

5. For thousand of example of my programming style, see leoPy.leo and leoGuiPlugins.leo. The projects section in leoPy.leo contains many examples of using clones to create view nodes. I typically delete the clones in the views shortly before a release.

Writing documentation

- Use postings as pre-writing for documentation.

  I don't mind blabbing on and on about Leo because all my posts become pre-writing for Leo's documentation. I simply copy posts to nodes in the ``documentation to-do" section. At release time, I edit these nodes and put them in Leo's main documentation or the release notes. This posting is an example.

- **Use the vr command to debug reStructuredText documentation. The viewrendered** pane updates as you type.
  This makes Leo a killer app for rST.

Administrative tips

- Never rely on memory.

  A project like this contains thousands and thousands of details. Everything eventually goes into a Leo node somewhere. If it doesn't it surely *will* be forgotten.

- Do easy items first.

  This keeps to-do lists short, which keeps energy high.

Tips for using bzr

I use the following batch files related to bzr:

```
b:      short for bzr
b c:    short for bzr commit
bs:     short for bzr status
tr:     short for cd <path to trunk>
main:   short for cd <path to copy of trunk>
```

The ``main'' (copy) of the trunk is purely for handling bzr conflicts. If one happens I do this:

```
main
b pull
b merge ../trunk
b c -m "my commit message"
b push
```

If the merge goes well (it usually does), I do this to resolve the conflict:

```
tr
b pull
```

4.3.7 How does Leo handle clone conflicts?

Some people seem to think that it is difficult to understand how Leo handles ``clone wars'': differing values for a cloned nodes that appear in several external files. That's not true. The rule is:

```
**The last clone that Leo reads wins.**
```

That is, for any cloned node C, Leo takes the value of C.h and C.b to be the values specified by the last copy that Leo reads.

There is only one complication:

```
**Leo reads the entire outline before reading any external files.**
```

Thus, if C appears in x.leo, y.py and z.py, Leo will choose the value for C in x.py or y.py, depending on which @<file> node appears later in the outline.

Note: Whenever Leo detects multiple values for C when opening an outline, Leo creates a ``Recovered nodes'' tree. This tree contains all the various values for C, nicely formatted so that it is easy to determine where the differences are.

4.4 Leo in Shared environments

4.4.1 How should I use Leo with bzr/git/hg/svn/cvs?

Using @file trees can eliminate most problems with using Leo in cooperative (SCCS) environments:

- Developers should use @file trees to create external files in any kind of cooperative environment.
- If sentinels are frowned upon in your development community, use @auto or @shadow instead of @file.
- The repository contains **reference** .leo files. These reference files should contain nothing but @file nodes. Reference files should change only when new external files get added to the project. Leo's bzr repository and Leo distributions contain the following reference files: LeoPyRef.leo, LeoPluginsRef.leo and leoGuiPluginsRef.leo. Developers should use local copies of reference files for their own work. For example, instead of using LeoPyRef.leo directly, I use a copy called LeoPy.leo.

4.4.2 How can I use Leo cooperatively without sentinels?

Leo's sentinels add outline structure to source files. However, those sentinels annoy some people who don't use Leo.

You can use @auto, @shadow or @nosent trees to edit files that are shared with those who don't want to see sentinel comments.

- @auto is best for files whose imported outline structure often changes. In most cases, this will be the best option. The drawback of @auto files are a) you can't use clones and b) you can't add your own organizer nodes.
- @shadow will work for files whose outline structure seldom changes. The advantage of @shadow is that you can add your own structure.
- @nosent is appropriate only for files that you alone modify.

4.4.3 What's the recommended way to upgrade Leo?

1. Archive and remove the previous version of Leo.
2. Download the nightly snapshot zip file.
3. Unzip it into the same place as the previous version.
4. Enjoy your up-to-date Leo code...

To make this work, it's important to keep your folder containing Leo separate from your .mySettings.leo and any data files.

4.5 Using external files

4.5.1 How do I inhibit sentinels in external files?

You have two options, depending on whether you want to be able to use sections or not.

- Use @nosent trees. Files derived from @nosent trees contain no sentinels. However, Leo create the external file just as in @file trees. In particular, Leo expands section references and understands the @others directive.
- Use @asis trees. Files derived from @asis trees contain no sentinels. Moreover, Leo does not expand section references in asis trees. In other words, Leo creates the *external file* simply by writing all body text in outline order. Leo can't update the outline unless the external file contains sentinels, so Leo does not update @nosent trees or @asis trees automatically when you change the external file in an external editor.

4.5.2 How do I prevent Leo from expanding sections?

Use @asis trees. Files derived from @asis trees contain no sentinels. Leo creates the external file simply by writing all body text in outline order. Leo can't update the outline unless the external file contains sentinels, so Leo does not update @asis trees automatically when you change the external file in an external editor.

4.5.3 How can I create Javascript comments?

Question: I'm writing a Windows Script Component, which is an XML file with a CData section containing javascript. I can get the XML as I want it by using @language html, but how can I get the tangling comments inside the CData section to be java-style comments rather than html ones?

Answer: In @file trees you use the @delims directive to change comment delimiters. For example:

```
@delims /* */
Javascript stuff
@delims <-- -->
HTML stuff
```

Important: Leo can not revert to previous delimiters automatically; you must change back to previous delimiters using another @delims directive.

4.5.4 How can I disable PHP comments?

By Zvi Boshernitzan: I was having trouble disabling `<?php' with comments (and couldn't override the comment character for the start of the page). Finally, I found a solution that worked, using php's heredoc string syntax:

```
@first <?php
@first $comment = <<<EOD
EOD;

// php code goes here.
echo "boogie";

$comment2 = <<<EOD
@last EOD;
@last ?>
```

or:

```
@first <?php
@first /*
*/

echo "hi";

@delims /* */
@last ?>
```

4.5.5 How can I use Leo with unsupported languages?

Here is a posting which might be helpful: http://sourceforge.net/forum/message.php?msg_id=2300457 The @first directive is the key to output usable code in unsupported languages. For example, to use Leo with the Basic language, use the following:

```
@first $IFDEF LEOHEADER
@delims '
@c
$ENDIF
```

So this would enable a basic compiler to ``jump" over the ``true" LEO-header-lines. Like this:

```
$IFDEF LEOHEADER <-conditional compilation directive
#@+leo-ver=4 <-these lines not compiled
#@+node:@file QParser005.INC
#@@first
#@delims '
'@@c
$ENDIF <-... Until here!
<rest of derived code file ... >
```

This changes the comment symbol the apostrophe, making comments parseable by a BASIC (or other language.)

4.5.6 How do I make external files start with a shebang line?

Use the @first directive in @file trees or @nosent trees.

The @first directive puts lines at the very start of files derived from @file. For example, the body text of @file spam.py might be:

```
@first #! /usr/bin/env python
```

The body text of @file foo.pl might be:

```
@first #/usr/bin/perl
```

Leo recognizes the @first directive only at the start of the body text of @file nodes. No text may precede @first directives. More than one @first directive may exist, like this:

```
@first #! /usr/bin/env python
@first # more comments.
```

4.5.7 Can @file trees contain material not in the external file?

No. Everything in an @file trees must be part of the external file: orphan and @ignore nodes are invalid in @file trees. This restriction should not be troublesome. For example, you can organize your outline like this:

```
+ myClass
..+ ignored stuff
..+ @file myClass
```

(As usual, + denotes a headline.) So you simply create a new node, called myClass, that holds your @file trees and stuff you don't want in the @file trees.

4.5.8 How can I use Leo with older C compilers

By Rich Ries. Some older C compilers don't understand the ``//'' comment symbol, so using @language C won't work. Moreover, the following does not always work either:

```
@comment /* */
```

This generates the following sentinel line:

```
/*@@comment /* */*/
```

in the output file, and not all C compilers allow nested comments, so the last */ generates an error. The solution is to use:

```
#if 0
@comment /* */
#endif
```

Leo is happy: it recognizes the @comment directive. The C compiler is happy: the C preprocessor strips out the offending line before the C compiler gets it.

4.5.9 Why can't I use @ignore directives in @file trees?

The @ignore directive can not be used elsewhere in @file trees because of the way Leo recreates outlines from external files. This is an absolutely crucial restriction and will never go away. For a few more details, see Leo 4.0: Eliminating error `recovery' in History of Leo.

There are several workaround, as shown in LeoPy.leo:

- keep notes in the outline outside of any external file.
- Use @all to gather notes in a external file, as in done in @file leoProjects.txt.

4.5.10 How can I avoid getting long lines in external files?

Question: I must follow a coding standard when writing source code. It includes a maximum line length restriction. How can I know the length of a line when it gets written to the external file?

Answer: If a node belongs to a external file hierarchy, its body might get indented when it is written to the external file. It happens when an @others directive or a section name appears indented in a higher-level node body. While (**line**, **col**) in status area show the line and column containing the body text's cursor, **fcol** shows the cursor coordinate relative to the external file, not to the current node. The relation **fcol >= col** is always true.

4.6 Customizing Leo

4.6.1 How can I add support for a new language?

See the instructions are in LeoPy.leo in:

```
Notes:How To:How to add support for a new language section.
```

This section contains clones of all relevant parts of Leo that you will change. Coming in Leo 4.4: Leo will use JEdit's language description files to drive the syntax colorer. To add support for a new language, just add another such description file.

4.6.2 How do I submit a plugin?

You have two options:

- Get cvs write access, and add the @file file to the plugins directory.
- Just send the @file file to me at edreamleo@gmail.com. That's all you need to do. In particular that there is no need to change leoPlugins.leo.

4.6.3 How do I add a new menu item from a plugin?

c.frame.menu.createMenuItemsFromTable will append items to the end of an existing menu. For example, the following script will add a new item at the end of the `File' menu:

```
def callback(*args,**keys):
    g.trace()

table = (("Test1",None,callback),)

c.frame.menu.createMenuItemsFromTable('File',table)
```

Plugins can do anything with menus using c.frame.menu.getMenu. For example, here is a script that adds a Test menu item after the `Open With' menu item in the File menu:

```
def callback(*args,**keys):
    g.trace()

fileMenu = c.frame.menu.getMenu('File')

# 3 is the position in the menu.  Other kinds of indices are possible.
fileMenu.insert(3,'command',label='Test2',command=callback)
```

4.6.4 How can I use Leo's legacy key bindings?

You can `revert' to old key bindings as follows:

1. Open leoSettings.leo.
2. Find the node `Keyboard shortcuts'.
3. Disable the old bindings by moving the node `@keys EKR bindings: Emacs keys + modes' so that it is a child of the node: `@ignore Unused key bindings'.
4. Notice that there are two child nodes of the node `@ignore Unused key bindings' that refer to legacy key bindings:
    - `@keys Legacy Leo shortcuts with important Emacs bindings'
    - `@keys Legacy Leo bindings'.
5. Move **one** of these two legacy nodes up one level so that it is a child of the node `Keyboard shortcuts'. It should **not** be a child of the node `@ignore Unused key bindings'.

4.6.5 How can I enable and disable support for psyco?

Find the @file leoApp.py node in leoPy.leo. In the ctor for the LeoApp class set self.use_psyco to True or False. You will find this ctor in the node:

```
Code-->Core classes...-->@file leoApp.py-->app.__init__
```

Note that this ivar can not be set using settings in leoSettings.leo because Leo uses g.app.use_psyco before processing configuration settings.

4.6.6 How do I specify qt fonts?

When using the Qt gui, you specify fonts using the node in leoSettings.leo called:

```
@data qt-gui-plugin-style-sheet
```

As usual, you will probably want to put this node in your myLeoSettings.leo file.

4.6.7 How do I set selection colors?

You set most colors in the following settings node:

```
@data qt-gui-plugin-style-sheet
```

However, settings for colors that can change during Leo's execution are found in the node:

```
Body pane colors
```

These settings are as follows, with the defaults as shown:

```
\@color body_cursor_background_color = None
\@color body_cursor_foreground_color = None
\@color body_insertion_cursor_color = None
\@color body_text_background_color = None
\@color body_text_foreground_color = None
\@color command_mode_bg_color = #f2fdff</vh></v>
\@color command_mode_fg_color = None</vh></v>
\@color insert_mode_bg_color = #fdf5f5</vh></v>
\@color insert_mode_fg_color = black</vh></v>
\@color overwrite_mode_bg_color = azure2</vh></v>
\@color overwrite_mode_fg_color = black</vh></v>
\@color unselected_body_bg_color = #ffffef</vh></v>
\@color unselected_body_fg_color = black</vh></v>
```

4.7 Tips and techniques

4.7.1 What is an easy way to profile code?

I had a need to figure out why a part of some python code I had written was taking too long.

I pulled the code into LEO and the relevant part of the outline looked something like this:

```
+ Main module
-- Generate cryptographic key
-- Hashing algorithm
```

etc. So I cloned just the segment I wanted to profile and pulled it under a new section:

```
+ Main module
-- [clone] Generate cryptographic key
-- Hashing algorithm

+ Profiling Experiment
-- [clone] Generate cryptographic key
```

And in the body of the ``Profiling experiment", I used this code:

```
code_under_here = """
@others
"""


from timeit import Timer
```

```
t = Timer("print my_key_generator()", code_under_here)
print t.timeit(number = 10)
```

And then I hit Control-B to execute the Profiling Experiment body. This let me make adjustments to the code in the clone body and keep hitting Control-B to execute the code with the timeit module to see immediately if what I had done was making a difference.

The great thing about this was that I just used the LEO @others construct to create a wrapper around the code and did not need to litter my code with debug or profiling statements. -- Kayvan

4.7.2 How can I do a simple find and replace?

The `official' way to start a replace command is:

```
<Ctrl-shift-r>find-pattern<return>replace-text<return>
```

But suppose you with start with:

```
<ctrl-f>find-pattern
```

and then realize you want to do a replace instead of a find. No problem. The following also works:

```
<Ctrl-f>find-pattern<Ctrl-shift-r>replace-text<return>
```

In other words, you can think of *<ctrl-f>* as meaning `show the find dialog'. There is another trick you should know. After typing *<ctrl-f>* or *<shift-ctrl-r>* you can use *<alt-ctrl>* keys to set or clear find options. For example:

```
<ctrl-f><alt-ctrl-w><find-pattern><return>
```

That is, *<ctrl-f>`shows the find dialog, `<alt-ctrl-w>* toggles the Whole Word checkbox and *<return>* starts the search. You can type the *<alt-ctrl>* keys anytime after *<ctrl-f>* (or *<shift-ctrl-r>*) and before *<return>*. You can also type multiple *<alt-ctrl-keys>* to toggle multiple checkboxes.

4.7.3 How can I use Leo to develop Leo itself?

The trick is to create a workflow that separates editing from testing. Putting test code in LeoPy.leo would waste a lot of time. To run tests you would have to exit Leo and reload LeoPy.leo. A much quicker way is to put all test code in a test.leo file. So to change and test code, do the following:

1. Save LeoPy.leo but do **not** exit Leo.
2. Quit the copy of Leo running test.leo, then reload test.leo.
3. Run test scripts from test.leo.

That's all. Python will recompile any changed .py files in the new copy of Leo. **Note**: I create a batch file called t.bat that runs test.leo, so to the ``edit-reload-test" cycle is just:

1. Control-S (in LeoPy.leo: saves the .leo file)
2. t (in a console window: runs test.leo, compiling all changed .py files as a side effect)
3. Control-E (in test.leo: runs the test script)

The benefits of the new workflow:

- test.leo loads _much_ more quickly than LeoPy.leo does. This new approach can increase the speed of the edit-reload-test cycle by more than a factor of 10. Hitting Control-S, t, Control-E takes about 5 seconds.
- LeoPy.leo runs with the *old* code, so it is much easier to fix syntax errors or exceptions in the *new* code: just fix the problem and save LeoPy.leo *without* closing LeoPy.leo, then restart test.leo. You run your tests on the new code, but you edit the new code with the old, stable code.
- test.leo is the perfect place to develop test. I can create and organize those tests and when I am done, `'test.leo" is a log of my work.

4.7.4 How can I import many files at once?

The Import Files dialog allows you to select multiple files provided you are running Python 2.3 or above. There is also an importFiles script in LeoPy.leo. You can use that script as follows:

```
import leo.core.leoImport as leoImport
leoImport.importFiles(aDirectory, ".py")
```

This will import all .py files from aDirectory, which should be a full path to a particular directory. You could use ".c" to import all .c files, etc.

4.7.5 How can I use two copies of Leo to advantage?

By Rich Ries. I often rework C code that's already been ``Leo-ized"--the first pass was quick and dirty to get it going. When I do subsequent passes, I wind up with subnodes that are out of order with the sequence found in the main node. It's not a big deal, but I like `em ordered. With just one editor pane, clicking on the node to move would switch focus to that node. I'd then need to re-focus on the main node. A minor nuisance, but it does slow you down.

My solution is to open a second editor with its focus on the main node. Switch to the other editor, and, referring to the first editor pane, move the nodes as you like. The second editor's pane will change focus to the node you're moving, but the first editor will stay focused on the main node. It's a lot easier to do than to describe!

4.7.6 How can I display graphics in Leo?

One way is to link directly to the media file from a Leo node (with @url) and write a script button to wrap all URL-nodes under the current node in a single HTML page. Then, you can view your media in two ways:

- Individually. You can directly click on the @url link to display the media in the browser (assuming you have your MIME/filetype associations set up correctly for your browser).
- In a group. You can click on a script button (you have to code this yourself, very simple) which should collect all @url nodes under the current node and dynamically generate a HTML page displaying either links to or embedded versions of the media (using the HTML trick described above to invoke the browser). This way, you can create collections of @url nodes under a single node (like a bookmark folder), and press a single button to view the @url collection as a single entity in the browser (with all browser capabilities like displaying the media).

You could probably generalize this idea of ``collect all @url nodes under current node and display as HTML in browser'' into a general-purpose plugin. However, the plugin would have to be somewhat smart in mapping a link to its corresponding HTML code (e.g. an image link gets mapped to an <img> HTML tag, a link to a Flash file gets mapped to an <embed> tag, etc).

4.7.7 How can I create a template .leo file?

Question: It would be nice if Leo could open empty files. I tend to be ``document oriented'' rather than ``application oriented'' in my thinking and prefer ``create empty file at location -> open it with program'' to ``start program -> create new file -> save it at location''.

Answer by Paul Paterson: If you are on Windows 98/2000/XP then the procedure is as follows...

1. Start Leo
2. Click New
3. Click Save as...
4. Save the file as ``c:\windows\shellnew\leofile.leo'' (or c:\winnt for 2000/XP)
5. Open regedit ``start...run...regedit''
6. Open HKEY_CLASSES_ROOT and find the ".leo" extension type
7. Go New ... Key from the context menu
8. Call the new key ShellNew
9. Select the new key, right-click, choose New...String Value from the context menu
10. Call it FileName
11. Double-click on the string, and modify it to be the filename of the leofile.leo file you created, including the extension
12. Exit the registry editor and restart Windows Explorer (you may need to reboot on Windows 98)

Now you should have a New:Leo File option in Explorer. This creates a duplicate of the file you saved. This can be useful because you could make a template Leo file containing some standard nodes that you always have and then save this.

4.7.8 How can I show Leo files with Excel?

From: http://sourceforge.net/forum/message.php?msg_id=3240374 Using Leo's File-Export-Flatten Outline commands creates a MORE style outline which places all Leo body sections on the left margin. The headlines are indented with tabs which Excel will read as a tab delimited format. Once inside Excel there are benefits.

1. The most obvious benefit inside Excel is that the body sections (Excel first column) can be selected easily and highlighted with a different font color. This makes the MORE format very readable. Save a copy of your sheet as HTML and now you have a web page with the body sections highlighted.
2. It is possible to hide columns in Excel. Hiding the first column leaves just the headlines showing.
3. Formulas based on searching for a string can do calculations in Excel. For example if a heading ``Current Assets'' appears on level 4 then the body formula:

```
=INDEX(A:A,MATCH("Current Assets",D:D,0)+1)
```

> will retrieve it. The +1 after match looks down one row below the matched headline. The trick is to place all your headlines in quotes because Excel will see + ``Current Assets" from the MORE outline. When Excel tries without the quotes it thinks it is a range name and displays a #N/A error instead of the headline. Also you must place a child node below to get the + sign instead of a - sign which would give a MORE headline of -``Current assets" , also is an error.

I think there is some interesting possibility here because of the enforcement of Leo body text being always in the first column. The Leo outline provides additional reference to organizing the problem not typical of spreadsheet models. Beyond scripting in Python, Excel is good at doing interrelated calculations and detecting problems like circular references. In Excel Tools-Options-General is a setting for r1c1 format which then shows numbers instead of letters for column references. Using this would allow entries like this in the leo body:

```
1000
3500
=R[-1]C+R[-2]C
```

In Excel you would see 4500 below those two numbers. This is completely independent of where the block of three cells exists on the sheet.

4.7.9 How can I reuse @button nodes in multiple files?

By Rich Ries

There is no direct way to make script buttons available in multiple Leo files. Sure, you could copy and paste the @button nodes, but there is a slightly easier way using the ``New Buttons" plugin.

1. Create and test and debug your desired Script Button.
2. With the Script Button node selected, run Plugins --> New buttons --> Make Template From

Open a new Leo file.

3. Assuming you have only the one New Button Template defined, left-click the New button, and a new node will be added to your outline. (Otherwise, you'll need to select the Template you want.)
4. Press [Script Button] to create the new script button.

It's easier to *do* this than to *explain* it!

4.7.10 How can I restore focus without using the mouse

It sometimes happens that the focus gets left in a Leo widget that doesn't support Leo's key bindings. You would think that you would have to use the mouse to click in, say, the body pane so that you can use Leo's key bindings again.

But you don't have to do that. Instead, use Alt-tab once to change away from Leo, and then use Alt-tab again to change back to Leo. When you do this, Leo puts focus in the body pane and you are all set.

4.7.11 How can I make commonly-used scripts widely accessible?

Put @command nodes as children of an @commands node in myLeoSettings.leo. This makes the the @command nodes available to all opened .leo files.

Using @command rather than @button means that there is never any need to disable scripts. There is no need for @button. To see the list of your @command nodes, type:

```
<alt-x>@c<tab>
```

Similarly to see the list of your @command nodes, type:

```
<alt-x>@b<tab>
```

4.7.12 How can I use BibTeX citations from Leo?

When using LaTeX and BibTeX, I would like to use inside of Leo a kind of LaTeX-inline-markup, that after generation of the RsT file through Sphinx as well as after running of ``make latex", generate a LaTeX file containing the citation call of the form cite{CITBook001} as described in a file *.bib. Is there a way to have Leo/Sphinx/RsT generate the inline raw latex syntax?

Use the docutils raw-data syntax. Examples:

```
.. role:: raw-role(raw)
  :format: html latex
.. raw:: latex
  \bibliographystyle{acm}
  \bibliography{myBibliography}
```

For more details, see this posting about BibTeX citations.

4.8 Trouble shooting

4.8.1 How do I get help?

All questions are welcome at http://groups.google.com/group/leo-editor

4.8.2 How do I report bugs?

You can discuss possible bugs at http://groups.google.com/group/leo-editor

Please report bugs at http://bugs.launchpad.net/leo-editor

When reporting a bug, please include *all* of the following:

- The version of Leo used.
- The version of Python used.
- The platform or platforms used: Linux, Windows, MacOS.

- A clear description of the problem.
- Information sufficient to recreate the problem.

It's polite to make the bug report self contained, so that six weeks later somebody will be able to understand the report as it stands.

4.8.3 My old .leo files won't load using Leo 4.5 or later. What should I do?

In version 4.5, Leo changed to using a sax parser for .leo files. This can cause problems if your .leo file contains invalid characters. Bugs in previous versions of Leo permitted these bad characters to appear.

The sax parser complains that these characters are not valid in .xml files. Remove these invalid characters as follows:

1. run Leo in a console window, and load the .leo file. Near the bottom of the error message you will see a line like:

```
SAXParseException: <unknown>:123:25: not well-formed (invalid token)
```

   This line reports a bad character at character 25 of line 123.

2. Open the .leo file in an external editor. The Scite editor, http://www.scintilla.org/SciTE.html, is a good choice because it clearly shows non-printing characters. Remove the invalid character, save the .leo file.

Repeat steps 1 and 2 until all invalid characters are gone.

4.8.4 Error messages from the rst3 plugin aren't helpful. What can I do?

For the most part, docutils does a good job of reporting errors. docutils prints a message to the console and inserts an unmistakable error message in the generated .html file. **Important**: On Windows it is helpful to run Leo in a console window.

However, in some cases, docutils crashes instead of properly reporting the problem. There are several workarounds:

1. The crashes I have seen arise from the following bug in docutils. **Hyperlinks in image:: markup must be lower case**. This will work:

```
.. .. |back| image:: arrow_lt.gif
    :target: faq_
```

   This will **crash**:

```
.. .. |back| image:: arrow_lt.gif
    :target: FAQ_
```

   So avoid this crash by making sure to use lower case targets in `:target:' markup.

2. You can change the docutils source slightly so that it prints a traceback when it crashes. (The rst3 plugin should be able to do this, but I haven't figured out how yet.) It's easy enough to do this:

   - Find the file core.py in top-level docutils folder. Typically this folder will be in Python's site-packages folder.
   - Open core.py in some editor other than Leo.
   - Find the method called report_Exceptions.

  - Insert the following lines at the very start of this method:

```
print 'EKR: added traceback'
import traceback ; traceback.print_exc()
```

  This will cause a traceback whenever docutils crashes. I have found that such tracebacks are generally enough to locate the general area of the problem. **Note**: These tracebacks go to the console window, so you should run Leo in a console window.

3. As a last resort, you can isolate syntax errors by reducing your input files until they work again, then adding sections until you get a crash. This is easy enough to do (when using the rst3 plugin) by change a headline `x' to @rst-ignore-tree x.

4.8.5 How can I run Leo from a console window?

Leo (and other programs) often send more detailed error messages to stderr, the output stream that goes to the console window. In Linux and MacOS environments, python programs normally execute with the console window visible. On Windows, can run Leo with the console window visible by associating .leo files with python.exe *not* pythonw.exe.

4.8.6 How can I use Python's pdb debugger with Leo?

Just run Leo in a console window. At the point you want to drop into the debugger, execute this line:

```
g.pdb()
```

All output from pdb goes to stdout, which is the console window. It would be good to create a subclass of pdb.Pdb that uses Leo's log pane rather than a console window, but I haven't done that. It could be done easily enough in a plugin...

Important: I recommend using g.trace instead of pdb. For example:

```
g.trace(x)
```

prints the name of the function or method containing the trace, and the value of x. g.callers is often useful in combination with g.trace. g.callers(5) returns the last 5 entries of the call stack. For example:

```
g.trace(x,g.callers(5))
```

Used this way, g.trace shows you patterns that will be invisible using pdb.

4.8.7 I can't write Imported files. What's going on?

The import commands insert @ignore directives in the top-level node. Leo does this so that you won't accidentally overwrite your files after importing them. Change the filename following @file (or @file) as desired, then remove the @ignore directive. Saving the outline will then create the external file.

4.8.8 Nothing (or almost nothing) happens when I start Leo. What should I do?

Missing modules can cause installation problems. If the installer doesn't work (or puts up a dialog containing no text), you may install Leo from the .zip file as described at How to install Leo on Windows. However you are installing Leo, be sure to run Leo in a console window. because as a last resort Leo prints error messages to the console.

4.8.9 The new Python decorator syntax causes problems. What can I do?

Python's decorator syntax is ill-conceived. This syntax file hack works well enough anyway to work with Leo `@' markup:

```
syn region leoComment start="^@\\s*" end="^@c\\s*$"
syn match   pythonDecorator "@\\S\\S+" display nextgroup=pythonFunction skipwhite
```

4.8.10 Running Python setup.py install from the leo directory doesn't work. Why not?

Leo's setup.py script is intended only to create source distributions. It can't be used to install Leo because Leo is not a Python package.

4.8.11 I can't run the LeoBridge module outside of leo/core. What should I do?

Question and answer from plumloco.

Add the equivalent of:

```
import sys
leocore = "path/to/leo/core"
if leocore not in sys.path: sys.path.append(leocore)
import leo.core.leoBridge as leoBridge
```

at the head of each file that uses leoBridge.

The problem is not importing leoBridge itself but (if I use `from leo.core') the importing of plugins, who get a different leoGlobals from leoBridge, without g.app etc, and so do not work if they rely on dynamic values in g.etc.

> Why can't you simply add leo/core to sys.path in sitecustomize.py?

Putting leo/core on the python path as you suggest would put forty python modules in the global module namespace for all python programs when I want just one. Also, I have a safe working copy of leo and a cvs/testing version. I would wish to test any programs against the testing version while using the working version, but both /core directories can't be exposed at the same time.

> Do you need plugins while running from the leoBridge?

Afraid so, at least the rst3 plugin. The solution I am using now is to place:

```
sys.modules['leoGlobals'] = leoGlobals
```

in leoBridge after import leo.core.leoGlobals as leoGlobals

This allows my scripts to be portable over the several computers/platforms I need to use them on, and makes testing scripts against multiple leo versions easy. It does mean that my scripts are not portable to other leo users but that is not likely to be a problem.

4.8.12 Why didn't Leo update my @shadow outline as expected?

As explained here, the fundamental @shadow algorithm guarantees *only* that writing an updated @shadow outline will generate the updated **public** file. There is *no way* to guarantee that the updated outline structure will be as expected. The @shadow algorithm can not *in principle* guess between two or more ways of updating the **private** file when each of the ways yields the same **public** file.

Happily, this ``fact of life" about @shadow is not serious. If you don't like the ``guesses" that the @shadow algorithm has made, you can simply change the @shadow tree. After saving the outline, the *private* file will record your choice. The next time you open the outline, you will see the choices *you* made, not the guesses that the @shadow algorithm made.

4.8.13 Why do Qt windows disappear in my scripts?

Q. When I run the following script I see a window appear and then immediately disappear:

```
from PyQt4 import QtGui
w = QtGui.QWidget()
w.resize(250, 150)
w.move(300, 300)
w.setWindowTitle('Simple test')
w.show()
```

What's going on?

A. When the script exits the sole reference to the window, w, ceases to exist, so the window is destroyed (garbage collected). To keep the window open, add the following code as the last line to keep the reference alive:

```
g.app.scriptsDict['my-script_w'] = w
```

Note that this reference will persist until the next time you run the execute-script. If you want something even more permanent, you can do something like:

```
g.app.my_script_w = w
```

4.9 Unicode issues

4.9.1 I can not enter non-ascii characters. What can I do?

Set @bool ignore_unbound_non_ascii_keys = False in LeoSettings.leo or myLeoSettings.leo.

4.9.2 Some characters in external files look funny. What can I do?

Internally, Leo represents all strings as unicode. Leo translates from a particular encoding to Unicode when reading .leo files or external files. Leo translates from Unicode to a particular encoding when writing external files. You may see strange looking characters if your text editor is expecting a different encoding. The encoding used in any external file is shown in the #@+leo sentinel line like this:

```
#@+leo-encoding=iso-8859-1.
```

Exception: the encoding is UTF-8 if no -encoding= field exists. You can also use the @encoding directive to set the encoding for individual external files. If no @encoding directive is in effect, Leo uses the following settings to translate to and from unicode:

default_derived_file_encoding The encoding used for external files if no @encoding directive is in effect. This setting also controls the encoding of files that Leo writes. The default is UTF-8 (case not important).

new_leo_file_encoding The encoding specified in the following line of new .leo files:

```
<?xml version="1.0" encoding="UTF-8">
```

The default is UTF-8 (upper case for compatibility for old versions of Leo).

4.9.3 I get weird results when defining unicode strings in scripts. What is going on?

Add the following to the start of your scripts:

```
@first # -*- coding: utf-8 -*-
```

Without this line, constructs such as:

```
u = u'a-(2 unicode characters here)-z'
u = 'a-(2 unicode characters here)-z'
```

will not work when executed with Leo's execute script command. Indeed, the Execute Script command creates the script by writing the tree containing the script to a string. This is done using Leo's write logic, and this logic converts the unicode input to a utf-8 encoded string. So *all non-ascii characters* get converted to their equivalent in the utf-8 encoding. Call these encoding <e1> and <e2>. In effect the script becomes:

```
u = u'a-<e1>-<e2>-z'
u = 'a-<e2>-<e>-z'
```

which is certainly *not* what the script writer intended! Rather than defining strings using actual characters, Instead, one should use the equivalent escape sequences. For example:

```
u = u'a-\\u0233-\\u8ce2-z'
u = 'a-\\u0233-\\u8ce2-z'
```

4.9.4 Some characters are garbled when importing files. What can I do?

The encoding used in the file being imported doesn't match the encoding in effect for Leo. You have two options:

- Use the @encoding directive in an ancestor of the node selected when doing the Import command to specify the encoding of file to be imported.

4.9.5 Python's print statement shows `byte hash' for unicode characters. What can I do?

First, you must change Python's default encoding to something other than `ascii'. To do this, put the following in your sitecustomize.py file in Python's Lib folder:

```
import sys
sys.setdefaultencoding('utf-8') # 'iso-8859-1' is another choice.
```

You must restart Python after doing this: sys.setdefaultencoding can not be called after Python starts up.

Leo's g.es_print and g.pr functions attempts to convert incoming arguments to unicode using the default encoding. For example, the following Leo script shows various ways of printing La Peoperly:

```
@first # -*- coding: utf-8 -*-

import sys
e = sys.getdefaultencoding()
print 'encoding',e
table = (
    'La Pe
    unicode('La Pe'utf-8'),
    u'La Pe
    u'La Pe\\xf1a',
)

for s in table:
    print type(s)
    g.es_print('g.es_print',s)
    if type(s) != type(u'a'):
        s = unicode(s,e)
    print 'print     ',s
    print 'repr(s)   ',repr(s)
```

For still more details, see: http://www.diveintopython.org/xml_processing/unicode.html

第 **5** 章

Slides

This is the front page for various slide shows about Leo.

5.1 Basic slide shows

Installation tells how to install Leo.

Leo Basics Step By Step explains the basics of Leo outlines.

External Files discusses creating external files with @file, @auto and @edit.

Clones and views illustrates how clones work and show how they create views.

Using Leo's Minibuffer tells how to execute Leo's commands by name.

5.2 Intermediate slide shows

Scripting Leo explains how to use Python scripting in Leo.

第 6 章

Installing Leo

This chapter tells how to install and run Leo on Windows or Linux. Leo can be installed on MacOS, but the process is difficult and not recommended.

Important: If you have *any* problems installing Leo, please ask for help on Leo's help forum.

Contents

- Installing Leo
  - Required and optional packages
  - Installing other packages
    - Installing Python
    - Installing Qt
    - Installing PyEnchant
  - Installing Leo itself
    - Installing Leo on Windows
    - Installing Leo on Linux
    - Installing Leo on MacOs 10.7 Lion
    - Installing Leo with bzr
  - Running Leo
    - Running Leo the first time
    - Running Leo in batch mode
    - Running Leo from a console window
    - The .leo directory

6.1 Required and optional packages

Leo requires Python. Leo will work on any platform that supports Python 2.6 or above, including Python 3.0 and above.

Leo also requires either the Qt widget set.

To enable spell checking, you must install the PyEnchant package.

The following sections describe how to install Python, Qt and PyEnchant.

6.2 Installing other packages

6.2.1 Installing Python

To install Python, see this page.

6.2.2 Installing Qt

To install Qt, get the binary package of PyQt from: http://www.riverbankcomputing.co.uk/software/pyqt/download

The version of PyQt that you download must match your installed Python version. Remember that Leo requires Python 2.6 or later, or Python 3.0 or later. Now run the binary PyQt installer.

6.2.3 Installing PyEnchant

You must install the PyEnchant package if you want to use Leo's Spell tab. Download and install the PyEnchant package from http://www.rfk.id.au/software/pyenchant/download.html There is an executable installer for Windows users.

6.3 Installing Leo itself

Leo's core code is always being improved and developed. Unit-testing ensures that the daily commits are as bug-free as possible. Almost all of the time, downloading the most recent nightly snapshot of the development code is going to give you code that is just as stable and much more up-to-date than the most recent latest stable release which most Leonistas would consider already outdated.

If you are just checking Leo out, feel free to use the latest stable release download if it makes you feel more secure, but once you've decided to work with Leo on a regular basis, we highly recommend regularly keeping your installation up to date with the most recent nightly snapshot.

To summarize, you may get Leo in three ways:

1. Download the latest stable release from SourceForge. This release contains an executable installer. This release will usually be a bit out of date.
2. Download a nightly snapshot from Leo's snapshots page. This page contains .zip archives of Leo's code from 1, 2, 5, 10, 30 and 90 days ago.
3. Download Leo's latest sources from Launchpad using bzr. Installing bzr is non-trivial, but once set up this is the easiest way to get the latest version of Leo's code.

6.3.1 Installing Leo on Windows

1. Install Python from http://www.python.org/download/releases/
2. Install Qt, as described above.
3. Download and install Leo. Download the latest version of Leo (a .zip file) from Leo's download page. You can unpack the .zip file anywhere, including Python's *site-packages* folder, for example, C:\Python26\Lib\site-packages

6.3.2 Installing Leo on Linux

If you are using Debian/Ubuntu, find and install the debian package. This provides the best integration with your desktop (file associations, icons, launcher item). Failing that, follow the instructions below.

Download the latest version of Leo (a .zip file) from Leo's download page.

Unzip the downloaded .zip file into the **unpacked folder** in your home directory. The unpacked folder will be called something like leo-4-5.

You now have two choices:

1. You can run Leo from your home directory. Just add ~/leo-4-5 to your path.
2. You can install leo into /usr/local/lib and /usr/local/bin by running Leo's install script as follows:

```
cd ~/leo-4-4-3-final
chmod u+x install
sudo ./install
```

The install script will instruct you to add /usr/local/bin to your path. You can, instead, add the following link:

```
sudo ln -s /usr/local/lib/leo/ /usr/local/lib/python2.6/site-packages/
```

Now you are ready to run Leo.

6.3.3 Installing Leo on MacOs 10.7 Lion

Important: Installing Leo on MacOS is, ah, challenging. Furthermore, Leo does not work as well on MacOS as on other platforms.

Many thanks to Ludwig Schwardt for the following installation instructions.

I recently received a new MacBook Pro and did a fresh upgrade to Mac OS 10.7 (Lion). I then used the opportunity to test out installation procedures of various software on a clean system. My main finding is that the excellent Homebrew (mxcl.github.com/homebrew/) makes things much easier these days.

Why Homebrew? It does not try to replace every single bit of functionality on your Mac with their own version, like Macports or fink. It reuses the existing libraries as far as possible. No need to reinstall Python, for example (one of my pet gripes when people try to install new software on their Macs, and the source of much confusion and pain). It installs to /usr/local, the standard place to find third-party libraries and headers, instead of the obscure /opt or /sw. It's simple to use and to extend.

I last installed Leo on Mac OS 10.4 (Tiger) back in the Tk days, and wondered what it looked like in Qt. All the horror stories of PyQT on Mac discouraged me from trying this before, so I was keen to see if Homebrew helps. Here is my installation write-up:

- Make sure you have Xcode installed (test it by confirming that ``gcc'' runs in the Terminal)
- In preparation for homebrew, the best option in my opinion is to delete /usr/local via:

```
sudo rm -rf /usr/local
```

  and install any software in it via homebrew instead. If this step fills you with dread and you do not want to lose your beloved third-party software, the second-best option is to make sure you have write permission for the directory via:

```
sudo chown -R <your user name>:admin /usr/local
```

  If you don't know your username, run ``whoami''. :-) This is useful because homebrew actually discourages you from installing third-party software as the superuser (the usual Mac apps in /Applications are also installed as the normal user, for that matter).

- Install Homebrew (http://mxcl.github.com/homebrew/) by running the following command in the Terminal:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.github.com/gist/323731)"
```

- Run ``brew update'' to get the latest formulas
- Now install PyQT (yes, that's it!):

```
brew install pyqt
```

- Run ``brew doctor'' and check any further suggestions to improve your system.
- Add the following lines to your ~/.bash_profile (or ~/.profile on Leopard):

```
export PATH=/usr/local/bin:$PATH
# This is for SIP (and PyQT) as suggested by Homebrew
export PYTHONPATH=/usr/local/lib/python:$PYTHONPATH
```

- Open a new Terminal tab / window so that the above settings take effect, and install Leo. I downloaded the Leo-4.9-final-a.zip, unzipped it, and ran ``python launchLeo.py'' inside the Leo directory.

We should consider adding a Homebrew formula for Leo. This will simplify the process even further, to simply ``brew install leo''. I started on this, but wasn't sure where to put the various Leo files in the system hierarchy. The Debian package can give some clues here, but I haven't looked at it yet.

6.3.4 Installing Leo with bzr

Many users will want to track the development version of Leo, in order to stay on top of the latest features and bug fixes. Running the development version is quite safe and easy, and it's also a requirement if you want to contribute to Leo.

1. First, you need to get Bazaar (bzr) from http://bazaar-vcs.org. For windows users we recommend the standalone installer - the python installer may have problems pushing to Launchpad. Plain bzr installer only contains the command line version, so you might want to augment that with a friendly GUI - qbzr is recommended as it's the easiest one to install. It provides command like bzr qlog, bzr qannotate etc.

2. Get Leo from launchpad by doing:

```
bzr branch lp:leo-editor
```

And that's it! You can run leo/core/leo.py directly. When you want to refresh the code with latest modifications from Launchpad, run bzr pull.

If you make modifications to Leo (with the interest in sharing them with the Leo community), you can check them in to your local branch by doing bzr checkin. Now, to actually request your changes to be merged to Leo trunk, you need a Launchpad account with RSA keys in place. There is showmedo video about how to accomplish this in Windows using puttygen and pageant at http://showmedo.com/videos/video?name=1510070&fromSeriesID=151.

After your Launchpad account is set up, go to https://launchpad.net/leo-editor, choose ``Code" tab -> Register Branch, select Branch type ``Hosted" and fill in descriptive details about the branch. After that, go to the branch home page from Code tab again, and copy-paste the push command line to terminal. For example, for branch:

```
https://code.launchpad.net/~leo-editor-team/leo-editor/mod_rclick
```

The push command is:

```
bzr push bzr+ssh://my_name@bazaar.launchpad.net/~leo-editor-team/leo-editor/mod_rclick
```

You may wish to add --remember command line option to bzr push, to direct all future pushes to that location. Then, you only need to execute bzr push.

After your branch is pushed, you can email the Leo mailing list and request it to be reviewed and merged to trunk.

6.4 Running Leo

You can run Leo from a Python interpreter as follows:

```
import leo
leo.run() # runs Leo, opening a new outline or,
leo.run(fileName=aFileName) # runs Leo, opening the given file name.
```

Another way to run Leo is as follows:

```
cd <path-to-launchLeo.py>
python launchLeo.py %*
```

Here are some tips that may make running Leo easier:

Linux The following shell script will allow you to open foo.leo files by typing leo foo:

```
#!/bin/sh
python <leopath>launchLeo.py $1
```

where <leopath> is the path to the directory containing the leo directory.

Windows You can associate Leo with .leo files using a batch file. Put the following .bat file in c:\Windows:

```
<path-to-python>/python <path-to-leo>/launchLeo.py %*
```

Here <path-to-leo> is the path to the directory *containing* the leo directory, that is, the directory containing launchLeo.py.

6.4.1 Running Leo the first time

The first time you start Leo, a dialog will ask you for a unique identifier. If you are using a source code control system such as bzr, use your bzr login name. Otherwise your initials will do.

Leo stores this identifier in the file .leoID.txt. Leo attempts to create leoID.txt in the .leo sub-directory of your home directory, then in Leo's config directory, and finally in Leo's core directory. You can change this identifier at any time by editing .leoID.txt.

6.4.2 Running Leo in batch mode

On startup, Leo looks for two arguments of the form:

```
--script scriptFile
```

If found, Leo enters batch mode. In batch mode Leo does not show any windows. Leo assumes the scriptFile contains a Python script and executes the contents of that file using Leo's Execute Script command. By default, Leo sends all output to the console window. Scripts in the scriptFile may disable or enable this output by calling app.log.disable or app.log.enable

Scripts in the scriptFile may execute any of Leo's commands except the Edit Body and Edit Headline commands. Those commands require interaction with the user. For example, the following batch script reads a Leo file and prints all the headlines in that file:

```python
path = r"<path-to-folder-containing-the-leo-folder>\\leo\\test\\test.leo"

g.app.log.disable() # disable reading messages while opening the file
flag,newFrame = g.openWithFileName(path,None)
g.app.log.enable() # re-enable the log.

for p in newFrame.c.all_positions():
  g.es(g.toEncodedString(p.h,"utf-8"))
```

6.4.3 Running Leo from a console window

Leo sends more detailed error messages to stderr, the output stream that goes to the console window. In Linux and MacOS environments, python programs normally execute with the console window visible. On Windows, can run Leo with the console window visible by associating .leo files with python.exe *not* pythonw.exe.

6.4.4 The .leo directory

Python's HOME environment variable specifies Leo's HOME directory. See http://docs.python.org/lib/os-procinfo.html for details.

Leo uses os.expanduser(`~') to determine the HOME directory if no HOME environment variable exists.

Leo puts several files in your HOME/.leo directory: .leoID.txt, .leoRecentFiles.txt, and myLeoSettings.leo.

第 **7** 章

The Leo Tutorial

Leo is a power tool for people who want to organize, study and work with data, especially complex data like computer programs, books, web sites and data bases. Superficially, Leo may look like other outlining programs, code folding editors or class browsers, but it most certainly is not.

People say Leo is a revolutionary tool, and that Leo is fun to use, even addictive. There is a unique ``Leo way'' of managing data; the term **Leonine** describes how people treat data in ``the world according to Leo''. Leo definitely takes a bit of work to understand. Leo's users speak of an ``Aha'' moment, when they see how these pieces fit together: outline structure is significant everywhere. For a more detailed introduction to Leo, see Leo in a nutshell.

Leo is freely available in source or binary form for all major platforms. You may download Leo from http://sourceforge.net/projects/leo/files/Leo/ Leo is Open Software and may be freely distributed.

Leo's home page contains additional documentation and links to other resources. For another introduction to Leo, open the file quickstart.leo in the leo/doc folder.

This tutorial introduces the reader to the basic concepts and features of Leo. It helps to have Leo running for hands-on experience, but all examples here are self-contained, so the tutorial can be read off-line as well. See Leo's Installation Guide. for detailed installation instructions. If you have problems installing Leo, please ask for help on Leo's forum.

This tutorial does not attempt to be comprehensive and cover every single feature of Leo, or even every commonly used feature. Instead, it introduces many of Leo's most noteworthy features, and will give you a good idea of Leo's flavor and style. After reading it, you will be able to use Leo in basic ways to create external files, organize data and run simple scripts. You will then be ready to learn more about Leo's many advanced features.

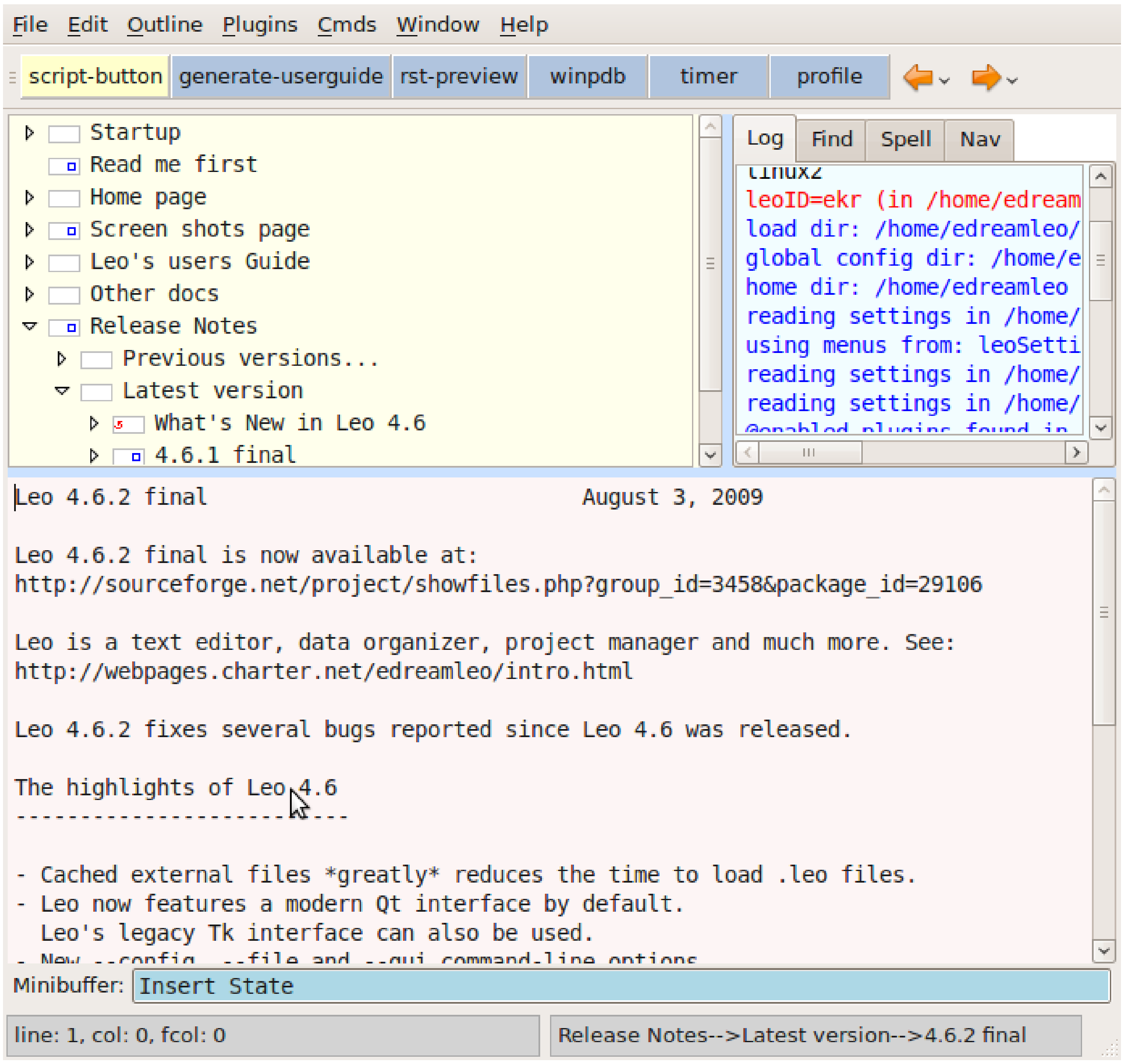
The Glossary is also worth reading.

Contents

- The Leo Tutorial
  - Leo's main window
  - External files and @file nodes
  - Creating external files from outlines
    - Section references
    - The @others directive
    - The @all directive
    - Choosing between @others and sections
    - Organizing programs as outlines
  - Clones & views
  - More about directives
  - Scripting, extending and customizing Leo
  - Summary
  - Further study

7.1 Leo's main window

Let's start looking at Leo in detail. We'll start with what you see when you first open Leo, Leo's main window. Leo's main window, shown below, represents an entire project. As you can see, the main window contains three panes: the **outline pane** at the top left, the **log pane** at the top right, and the **body pane** at the bottom. The window also contains an **icon area** at the very top, a **status area** and a **mini-buffer** at the very bottom.



```
linux2
leoID=ekr (in /home/edream
load dir: /home/edreamleo/
global config dir: /home/e
home dir: /home/edreamleo
reading settings in /home/
using menus from: leoSetti
reading settings in /home/
reading settings in /home/
@enabled plugins found in
```

```
Leo 4.6.2 final                        August 3, 2009

Leo 4.6.2 final is now available at:
http://sourceforge.net/project/showfiles.php?group_id=3458&package_id=29106

Leo is a text editor, data organizer, project manager and much more. See:
http://webpages.charter.net/edreamleo/intro.html

Leo 4.6.2 fixes several bugs reported since Leo 4.6 was released.

The highlights of Leo 4.6
--------------------------

- Cached external files *greatly* reduces the time to load .leo files.
- Leo now features a modern Qt interface by default.
  Leo's legacy Tk interface can also be used.
- New --config, --file and --gui command-line options
```

Outline pane

The outline pane shows your project as an outline. The outline contains all your project's data. An outline consists of **nodes**. The **icon box** is a small icon directly to the left of the headline text. The border of the icon box is black if the node has been changed. Smaller icons within the icon box indicate the status of the node:

A small blue box:	the node has body text.
A red vertical bar:	the node is marked.
A circular arrow:	the node is cloned.

If a node contains children, a smaller icon appears to the left of the icon box. This icon contains a `+' or `-` symbol. Clicking this **expansion box** expands or contracts the node.

Node

Each outline node has two two parts, a **headline** and **body text**. The outline pane shows headlines. Selecting a headline selects the entire node; the node's body text appears in the body pane. Leo uses standard terminology to describe the relationships of nodes in an outline. We speak of **parent** nodes, **child** nodes, **ancestor** nodes and **descendant** nodes.

Body pane

The body pane contains the body text of the node selected in the outline pane.

Log pane

The log pane contains informational messages from Leo or your scripts.

Icon area

Depending on what plugins are enabled, the icon area may contain buttons and other widgets that extend what Leo can do. The scripting plugin makes it easy to add buttons to the icon area.

Status area

The status area shows the line and column containing the body text's cursor, and the **UNL** (Uniform Node Location), the path from the top of the outline to the selected node. This path will change as you change outline nodes.

Minibuffer

You can type command and search strings in the minibuffer. It works much like the Emacs mini-buffer. To enter a command, type <Alt-x> followed by the command name and then <return>. To type a search string, type <ctrl-f> followed by the search string and then <return>. For full details, see Using Leo's Commands.

7.2 External files and @file nodes

Leo stores outline data on your file system in **.leo files**. The format of these files is XML. You don't have to store all your data in .leo files: Leo allows you to store parts of your outline data **external files**, that is, other files on your file system.

@file nodes create external files. @file nodes have headlines starting with @file followed by a file name. Some examples:

```
@file leoNodes.py
@file ../../notes.text
```

The file name can be an absolute path or a relative path to the file that starts at Leo's **load directory**, the directory containing the .leo file.

Leo reads and writes external files automatically when you open or save your Leo outline:

- When you open an outline (.leo file) Leo reads all the external files created by the @file nodes in the outline. If you have changed an external file outside of Leo, Leo will update the corresponding @file tree to reflect those changes when Leo next opens the outline.
- When you save your outline, Leo writes all **dirty** @file nodes. An @file is dirty if the node or any of its descendant nodes has changed. **Important**: When Leo writes an external file, Leo writes all the essential information in the @file tree to the external file, *not* to the .leo file. The only nodes that gets written to the .leo file are nodes that are not contained in any @file tree.

7.3 Creating external files from outlines

We come now to one of Leo's most important and unusual features. When Leo writes an external file, it does so in a flexible manner, directed by **outline-based markup**. This markup tells Leo exactly how to create the external file from an @file node.

The **obvious** way to write an external file would be to write the @file node itself followed by all the descendant nodes in **outline order** (the order in which nodes appear in the outline). But Leo does *not* write external files exactly this way.

Yes, Leo does indeed start by writing the @file node itself. But Leo writes the @file node's descendants only when it sees one of three kinds of Leo markup: section references, the @others directive and the @all directive. We'll discuss these three kinds of markup in the next section.

Section references and the @others and @all directives tell Leo to write the **expansion** of one or more descendant nodes to the external file. Programmers will recognize this process as akin to macro expansion. The following sections will explain this process in detail.

7.3.1 Section references

A **section reference** is a line of body text of the form:

```
<< a section name >>
```

Here, ``a section name'' can be any descriptive text not containing ``>>''. When Leo encounters a section reference, Leo searches all the descendants of the node containing the reference looking for a node whose headline matches the section reference. That is, Leo looks for a descendant node whose headline starts with:

```
<< a section name >>
```

We call such nodes **named nodes**. Leo doesn't require an exact match. Leo ignores whitespace and the case of letters when comparing headlines to section reference. Also, Leo ignores anything that may follow the section name in a named node. For example, the following headline will match the section reference above:

```
<< A Section Name >> (to do)
```

If Leo does find a match, Leo *replaces* the section reference (``<< a section name>>'') by the *expansion* of the body text of the matched node. That is, Leo replaces the section reference by the body text of the matched node, but Leo **expands all markup** in the matched node *before* making the replacement. The entire expansion of the matched node replaces the original section reference. Programmers will recognize this process as recursive macro expansion.

We have just discussed what happens if Leo does find a descendant named node that matches the section reference. If no such match is found the section reference is said to be **undefined** and Leo does not write any data to the external file. This is *not* a serious error: Leo will will save the erroneous @<file> tree in the .leo file instead of the external file. No information is lost. By the way, Leo's syntax coloring will indicate undefined section reference by underlining the section name.

Important: the indentation of section references matters. When expanding a section reference, Leo indents every line of the expansion by the leading whitespace that occurs before the section reference. Note also that you can't write something after a section reference and expect it to end up on the same line after expansion--Leo always writes a newline after the expansion.

7.3.2 The @others directive

The **@others directive** is the second (and most common) way of including descendant nodes in an external files. When Leo encounters the @others directive it replaces the @others directive by the *expansion* of all **unnamed** descendant nodes. As with section references, Leo replaces all markup in the descendant nodes, and the entire expansion replaces the @others directive.

In short, section references write *named* nodes; @others directives write all *unnamed* nodes. By the way, no node may contain more than one @others directive because there would be no way to ``apportion'' descendant nodes to more than one @others directive. However, nodes may contain as many section references as you like.

As with section references, the indentation of the @others directive matters. This allows Leo to handle Python source code properly. For example, the following is a common way of representing a Python class:

```
class myClass:
    '''a docstring'''
    @others
```

When Leo writes this node to an external file, Leo will write the first two lines to the external file, with the indentation in effect for the node. Leo will then write all descendant nodes to the external files, with *additional* indentation equal to the leading whitespace appearing before the @others directive.

7.3.3 The @all directive

The @all directive is the third, simplest (and least common) way of including descendant nodes. This directive causes Leo to write all descendant nodes in outline order, regardless of whether they are named or not. Furthermore, the @all directive does not expand any markup in descendant nodes. This results in Leo writing the external file in the ``obvious'' way. That is, Leo writes all descendant nodes in outline order.

Use the all directive if your external file contains unrelated nodes. For example, I use an external file to store programming notes. These notes typically contain snippets of programming source code, but there is no real relationships between the snippets--the file is simply a grab bag of information. The @all directive is designed for this situation.

7.3.4 Choosing between @others and sections

Newcomers to Leo frequently ask when to use the @others directive and when to use sections. It is good style to use section references only when the order of text within a external file matters. For example, Python programmers put docstrings and imports at the start of files. So the body text of @file nodes typically look something like this:

```
<< docstring >>
@language python
@tabwidth -4
<< imports >>
@others
```

This ensures that the docstring is first in the file, followed by imports, followed by everything else. Note that the order in which functions are defined in a file, or methods defined within a class, typically does *not* matter. Thus, it is good style to define classes like this:

```
class myClass:
    << class attributes >>
    @others
```

It would be bad style to define a class like this:

```
class myClass:
    << class attributes >>
    << method 1 >>
    << method 2 >>
    ...
```

Not only does this over-specify the order in which methods are defined, but it requires lots of extra typing. Not only must you add a line for each method, but headlines must contain section names such as << method 1 >>, <<method 2>>, etc. When using @others it is good style simply to put the name of each method in the headline.

7.3.5 Organizing programs as outlines

A few more words about style:

- It is good style to put each class, function or method in its own node. This makes it easy to see the shape of your code.
- It is good style to use organizer nodes to group related functions or methods. An organizer node has no content except maybe for comments. Like this:

  ```
  + myClass
    + birth and death
      + __init__
      etc.
    + getters
      etc.
    + setters
      etc.
    + misc methods
      etc.
  ```

  (In this notation, `+' denotes a headline.) This organization is far superior to using hideous comments like:

  ```
  ###########
  # Getters #
  ###########
  ```

- It is bad style to use @others in organizer nodes. There is no need to do so.
- It is bad style to use @others when order does matter. The reason is that it is very easy to move nodes in a tree by mistake, say by alphabetizing nodes. One wants to make the meaning of a external file immune from such movements.

One last word about style. The world won't end if you happen to use bad style by mistake: you just might cause a bit more work for yourself than was strictly necessary. Feel free to invent your own style of using Leo. Still, it would be wise to ``know the rules before you break them."

7.4 Clones & views

A **clone** is a node that appears in more than one place in a Leo outline. Clones are marked with a small red arrow in the icon box. All clones of a node are actually *the same node*, so any change to one clone affects all clones. For example, inserting, moving or deleting any child of a clone will change all other clones on the screen.

Please take a few moments to experiment with clones. Create a node whose headline is A. Clone node A using the Clone Node command in Leo's Outline menu. Type some text into the body of either clone of A. The same text appears in the bodies of all other clones of A. Now insert a node, say B, as a child of any of the A nodes. All the A nodes now have a B child. See what happens if you clone B. See what happens if you insert, delete or move nodes that are children of A. Verify that when you delete the penultimate clone, the last clone becomes a regular node again.

Clones are much more than a cute feature. Clones allow multiple views of data to exist **within a single outline**. With Leo, there is no such thing as a single, ``correct'' view of data. You can have as many views of data as you like.

To create a new view of the data in your outline, just do the following:

1. Create an *ordinary* node, that will represent the view. We call these nodes **view nodes** merely to indicate they represent a view.
2. Clone all the nodes from the outline that you want the view to contain. Move these clones so they become children of the view node.
3. (Optional) You can add regular nodes as children of the view node too.

For example, when I fix a bug in Leo, I create an ordinary node to represent the bug. This **bug node** is my view of all the data in Leo's source code that relates to the bug. As I discover code related to the bug, I clone their nodes and move them under the bug node. I'll also add ordinary nodes as children of the bug node. These nodes contain the original bug report, descriptions of how I fixed the bug, test data, or any other notes I might want to keep.

Once I have created the bug node, I concentrate *only* on that node and its children. I can examine the bug node and its children without having to jump around the outline. Everything I need is in one place. When I get around to actually fixing the bug I can do so by changing the clones. Again, I do not have to jump around the outline. It doesn't matter how big or complex the entire outline is: I am only dealing with the bug node and its children. This extremely narrow focus makes it *much* easier to fix bugs.

By the way, I never have to remember to save external files. When I change any clone, Leo marks all instances of that clone throughout the entire outline as dirty (changed). When I save the Leo outline, Leo automatically writes all the external files that contain dirty nodes.

Views have an unlimited number of uses. Use them whenever you want to focus your attention on some smaller set of nodes. For example, I often create view nodes when studying other people's code. The view node helps me concentrate on just the part of the code that interests me at the moment.

7.5 More about directives

Leo's **directives** control such things as syntax coloring, line wrapping within the body pane and the width of tabs. Leo directives may appear in headlines or body text. Leo directives start with `@', followed by the name of the directive.

Note: Leo handles Python decorators properly, providing they don't conflict with Leo's directives.

Here are some of Leo's directives:

```
@language python
@tabwidth -4
@wrap
@nowrap
@color
@nocolor
@killcolor
```

Most directives must start with the `@' in the leftmost column, but whitespace may appear before the `@others` and `@all` directives. As we have seen, such whitespace is significant.

Directives apply until overridden in a subtree. All of these directives apply to the node they are contained in, and also to the entire tree of descendant nodes, unless **over-ridden** by a similar directive in a descendant node. For example, the directive:

```
@language python
```

tells Leo to syntax color the node and all descendant nodes as Python code. However, some descendant node might contain:

```
@language rest
```

which tells Leo to color that node and all of *its* descendants as reStructureText. This principle applies to almost all of Leo's directives: the directive is in effect throughout a tree, unless overridden in some subtree.

@color, @nocolor and @killcolor

> These directives control how Leo colors body text. You can mix @nocolor and @color directives in a single node. This directives affect descendant nodes unless a node contains both @color and @color. Such **ambiguous** nodes do not affect the coloring of descendant nodes.

@first

> This directive forces a lines to appear before the first sentinel of a external file. Here is a common way to start a Python file:
>
> ```
> @first #! /usr/bin/env python
> @first # -*- coding: utf-8 -*-
> ```

@language

> Sets the language in effect for a tree. This affects how Leo colors body text. It also sets the comment delimiters used in external files. Leo supports dozens of languages. See *Leo's reference* for a complete list. Here are a few:
>
> ```
> @language python
> @language c
> @language rest # restructured text
> @language plain # plain text: no syntax coloring.
> ```

@pagewidth <n>

> Sets the page width used to format break doc:

```
@pagewidth 100
```

@path <path>

This directive is a convenience. Rather than specifying long paths in @file nodes, you can specify a path in an ancestor @path node. For example, suppose three nodes have the following headlines:

```
@path a
  @path b
    @file c/d.py
```

Because of the ancestor @path nodes, the @file node creates the file a/b/c/d.py

Within @path and @<file> paths, {{exp}} gets evaluated with the following symbols known: c, g, p, os and sys. For example:

```
@file {{os.path.abspath(os.curdir)}}/abc.py
```

refers to the file abc.py in (absolute path of) the current directory.

@tabwidth

Sets the width of tabs. Negative tab widths cause Leo to convert tabs to spaces and are highly recommended for Python programming.

@wrap and @nowrap.

These enable or disable line wrapping the Leo's body pane.

7.6 Scripting, extending and customizing Leo

Leo is fully scriptable using the Python language. Leo can execute any body text as a Python script. To run the entire body text as a script, simply choose the node and execute the Execute Script command (Ctrl+B). If text is selected, the Execute Script command will run just the selected text as the script.

The Execute Script command **preprocesses** the script before executing it, in exactly the same way that Leo writes external files. Leo expands section references and processes @others directives before executing the script. This allows you to use all of Leo's outlining capabilities to organize your scripts.

Your Python scripts can easily access data in an outline. Leo's execute-script (Ctrl-B) command predefines three variables, c, g and p, that scripts can use to easily access any part of any Leo outline, and Leo's own source code. For example, the following script will print all the headlines in an outline:

```
for p in c.all_positions():
  print(' '*p.level(),p.h)
```

The example above is only the beginning of what scripts can do. See Scripting Leo with Python for a complete discussion of scripting Leo.

Plugins are Python modules that change how Leo works. Leo's user have contributed dozens of plugins that have extended Leo's capabilities in many new directions. The file leoPlugins.leo contains all plugins that are included in Leo distributions.

Plugins and other parts of Leo can get options from **@settings** trees. @settings trees allow plugins to get options without any further support from Leo's core code. For a full discussion of @settings trees, see Customizing Leo.

7.7 Summary

Using Leo quickly becomes second nature:

- You can use Leo like any ordinary outliner, as a filing cabinet, but Leo's clones makes this filing cabinet much more flexible and useful than usual.
- You create external files using @file trees. Within @file trees, you use section references and the @others directive to tell Leo how to write nodes to the external file. Directives such as @tabwidth and @language provide other information to Leo. Leo's @file trees allow you to organize your scripts and programs with Leo's outline structure.
- You can execute Python scripts from any node in a Leo outline. Leo scripts have full, easy, access to all the information in the outline. Using scripts and plugins, you can easily add new features to Leo.

7.8 Further study

LeoPyRef.leo (in the core subdirectory of the leo folder) contains almost all of Leo's source code. It provides hundreds of examples of everything discussed here. This file will repay close study. For full details on all aspects of Leo see LeoDocs.leo.

第 **8** 章

Using Leo

This chapter discusses the basics of using Leo, including all of Leo's commands. It starts with a discussion of the Emacs-like minibuffer, then continues with a discussion of commands in each of Leo's menus.

Contents

- Using Leo
  - The minibuffer and minibuffer commands
    - Basic editing commands
    - Debugging commands
    - Emacs commands
    - Find commands
    - Gui commands
    - Help commands
    - Mode commands
    - Outline commands
    - Miscellaneous commands
  - Autocompletion
    - Starting autocompletion
    - Using the QCompleter
    - Using the Log pane completer
    - Showing docstrings
    - Autocompleter settings
    - Creating ctags data
  - Calltips
  - File commands
    - Loading, Saving and Reverting Files
    - Communicating with external editors
    - Importing Files into Leo Outlines
    - Exporting Files from Leo Outlines
    - Quitting Leo
  - Edit commands
    - Undoing changes
    - Cutting, pasting and selecting text
    - Indenting body text
    - Adding and deleting comments in body text
    - Creating nodes from body text
    - Converting leading blanks and tabs in body text
    - Executing Python scripts in body text
    - Finding and changing text
    - Go To Line Number
    - Inserting the date and time
    - Reformatting paragraphs in body text
    - Matching brackets and parenthesis
    - Indenting body text automatically
    - Creating and destroying multiple body editors
    - Opening URL's
    - Using chapters
  - Outline commands
    - Creating and destroying nodes
    - Expanding & contracting nodes
    - Cutting, pasting and deleting nodes
    - Navigating through the outline
    - Moving & Reorganizing nodes

8.1 The minibuffer and minibuffer commands

The mini-buffer is a text area at the bottom of the body pane. You use it like the Emacs mini-buffer to invoke commands by their so-called *long name*.

The full-command (Alt-x) command puts the focus in the minibuffer. Type a full command name, then hit <Return> to execute the command. Tab completion works, but not yet for file names. For example, to print a list of all commands type:

```
<Alt-X>print-commands<Return>
```

Extremely important: Like Emacs, many of Leo's commands have long-winded names. It is **not**, repeat **not** necessary to type the entire name in the minibuffer! Instead, you can use **tab completion** to shortcut your typing. For example, Suppose you want to execute the print-commands return. First, you type the first few characters of the command, and hit the <tab> key:

```
<Alt-x>pri<Tab>
```

You will see the following list of completions in the log window:

```
print-all-uas
print-bindings
print-cmd-docstrings
print-commands
print-focus
print-node-uas
print-plugin-handlers
print-plugins-info
print-settings
```

The minibuffer now contains the **longest common prefix** of all the completions, in this case:

```
print-
```

So now, all you have to do is type:

```
s<tab>
```

and the minibuffer will show:

```
print-settings
```

Finally, execute the command by hitting the <return> key. Using tab completion quickly becomes second nature. It saves a huge amount of typing. More importantly, it means that not every command needs to be bound to a keystroke in order to be conveniently available.

The keyboard-quit (Ctrl-g) commands exits any minibuffer mode and puts the focus in the body pane. **Important**: Use ctrl-g whenever you are unsure of what is happening.

The following sections list the various commands that you can invoke from the minibuffer. **Important**: you may bind keystrokes to any of these commands. See Customizing Leo for full details.

8.1.1 Basic editing commands

Here is a list of Leo's basic editing commands with the default (EKR) bindings shown:

Left	back-char
Shift-Left	back-char-extend-selection
	back-paragraph
	back-paragraph-extend-selection
	back-sentence
	back-sentence-extend-selection
	back-to-indentation
Alt-B or Ctrl-Left	back-word
Alt-Shift-B	back-word-extend-selection
Ctrl-Shift-Left	back-word-extend-selection
BackSpace	backward-delete-char
Shift-BackSpace	backward-delete-char
Ctrl-BackSpace	backward-delete-word
	backward-delete-word-smart
	backward-kill-paragraph
	backward-kill-sentence
	backward-kill-word
Ctrl-Home	beginning-of-buffer
Ctrl-Shift-Home	beginning-of-buffer-extend-selection
Home	beginning-of-line
Shift-Home	beginning-of-line-extend-selection
Ctrl-C	copy-text
Ctrl-X	cut-text
Delete	delete-char
Ctrl-End	end-of-buffer
Ctrl-Shift-End	end-of-buffer-extend-selection
End	end-of-line
Shift-End	end-of-line-extend-selection
Alt-M	exchange-point-mark
Ctrl-W	extend-to-word
Right	forward-char
Shift-Right	forward-char-extend-selection
	forward-paragraph
	forward-paragraph-extend-selection
	forward-sentence
	forward-sentence-extend-selection
Ctrl-Right	forward-word
Ctrl-Shift-Right	forward-word-extend-selection
Tab	indent-region
Return	insert-newline
	move-lines-down
	move-lines-up
	move-past-close
	move-past-close-extend-selection
Ctrl-J or Tab	newline-and-indent

```
Down            next-line
Shift-Down        next-line-extend-selection
Ctrl-V          paste-text
Up              previous-line
Shift-Up          previous-line-extend-selection
Shift-Ctrl-Z      redo
Ctrl-A          select-all
Ctrl-Z          undo
Shift-Tab         unindent-region
```

8.1.2 Debugging commands

These commands are for debugging Leo itself:

```
collect-garbage
debug
disable-gc-trace
dump-all-objects
dump-new-objects
dump-outline
enable-gc-trace
free-tree-widgets
print-focus
print-gc-summary
print-stats
verbose-dump-objects
```

8.1.3 Emacs commands

The following commands work just like their Emacs counterparts. Use the help-for-command command for further details:

```
        add-space-to-lines
        add-tab-to-lines
        advertised-undo
        capitalize-word
        center-line
        center-region
        clean-lines
        clear-extend-mode
        clear-kill-ring
        clear-rectangle
        clear-selected-text
        count-pages
        count-region
        dabbrev-completion
        dabbrev-expands
        delete-comments
```

	delete-file
	delete-indentation
	delete-rectangle
	delete-spaces
	diff
	digit-argument
	downcase-region
	downcase-word
	end-kbd-macro
	escape
	eval-expression
	expand-region-abbrevs
	fill-paragraph
	fill-region
	fill-region-as-paragraph
	flush-lines
	full-command
	goto-char
	how-many
	increment-register
	indent-region
	indent-relative
	indent-rigidly
	indent-to-comment-column
	insert-body-time
	insert-file
	insert-keyboard-macro
	insert-parentheses
	insert-register
	inverse-add-global-abbrev
	jump-to-register
	keep-lines
	keyboard-quit
	kill-all-abbrevs
	kill-buffer
Ctrl-K	kill-line
	kill-paragraph
	kill-rectangle
	kill-region
	kill-region-save
	kill-sentence
	kill-word
	line-number
	list-abbrevs
	list-buffers-alphabetically
	load-file
	macro-call
	macro-call-last

```
macro-end-recording
macro-load-all
macro-name-last
macro-print-all
macro-print-last
macro-save-all
macro-start-recording
make-directory
match-bracket
name-last-kbd-macro
negative-argument
number-command
number-command-0
number-command-1
number-command-2
number-command-3
number-command-4
number-command-5
number-command-6
number-command-7
number-command-8
number-command-9
open-rectangle
point-to-register
prepend-to-buffer
prepend-to-register
read-abbrev-file
rectangle-clear
rectangle-close
rectangle-delete
rectangle-kill
rectangle-open
rectangle-string
rectangle-yank
register-append-to
register-copy-rectangle-to
register-copy-to
register-increment
register-insert
register-jump-to
register-point-to
register-prepend-to
register-view
remove-blank-lines
remove-directory
remove-sentinels
remove-space-from-lines
remove-tab-from-lines
```

```
          rename-buffer
          repeat-complex-command
          reverse-region
          run-unit-tests
          select-paragraph
          set-comment-column
          set-fill-column
          set-fill-prefix
          shell-command
          shell-command-on-region
          sort-columns
          sort-fields
          sort-lines
          split-line
          start-kbd-macro
          string-rectangle
          suspend
          switch-to-buffer
          tabify
          transpose-chars
          transpose-lines
          transpose-words
          unindent-region
          universal-argument
          unmark-all
          untabify
          upcase-region
          upcase-word
          view-lossage
          what-line
Ctrl-Y    yank
Alt-Y     yank-pop
          zap-to-character
```

8.1.4 Find commands

Here is a list of all of Leo's find commands. The apropos-find-commands command will print a detailed help message discussing these commands:

```
             clone-find-all
             clone-find-all-flattened
             find-all
Ctrl+Key+2   find-character
Ctrl+Shift-2 find-character-extend-selection
             find-clone-all
             find-clone-all-flattened
F3           find-next
             find-next-clone
```

F2	find-prev
	find-quick
Ctrl+Shift+F	find-quick-selected
	find-quick-test-failures
	find-quick-timeline
	find-tab-hide
	find-tab-open
	find-word
	find-word-in-line
Alt+R	isearch-backward
	isearch-backward-regexp
Alt+S	isearch-forward
	isearch-forward-regexp
	isearch-with-present-options
	re-search-backward
	re-search-forward
	replace-string
	search-again
	search-backward
	search-forward
Ctrl-F	search-with-present-options
Alt+Ctrl+E	set-find-everywhere
Alt+Ctrl+N	set-find-node-only
Alt+Ctrl+S	set-find-suboutline-only
	toggle-find-collapses-nodes
Alt+Ctrl+I	toggle-find-ignore-case-option
Alt+Ctrl+B	toggle-find-in-body-option
Alt+Ctrl+H	toggle-find-in-headline-option
Alt+Ctrl+C	toggle-find-mark-changes-option
Alt+Ctrl+F	toggle-find-mark-finds-option
Alt+Ctrl+X	toggle-find-regex-option
Alt+Ctrl+W	toggle-find-word-option
Alt+Ctrl+A	toggle-find-wrap-around-option
	word-search-backward
	word-search-forward

8.1.5 Gui commands

The following commands simulate mouse clicks, double-clicks or drags:

	abort-edit-headline
	activate-cmds-menu
	activate-edit-menu
	activate-file-menu
	activate-help-menu
	activate-outline-menu
	activate-plugins-menu
	activate-window-menu

	add-editor
	cascade-windows
	click-click-box
	click-headline
	click-icon-box
	close-window
	contract-body-pane
	contract-log-pane
	contract-outline-pane
Alt-Ctrl-Minus	contract-pane
Ctrl-Shift-Tab	cycle-all-focus
	cycle-editor-focus
	cycle-focus
	delete-editor
	double-click-headline
	double-click-icon-box
	edit-headline
	end-edit-headline
	equal-sized-panes
	expand-body-pane
	expand-log-pane
	expand-outline-pane
Alt-Ctrl-Plus	expand-pane
Alt-D	focus-to-body
	focus-to-log
	focus-to-minibuffer
Alt-T	focus-to-tree
	fully-expand-body-pane
	fully-expand-log-pane
	fully-expand-outline-pane
	fully-expand-pane
	hide-body-pane
	hide-find-tab
	hide-invisibles
	hide-log-pane
	hide-outline-pane
	hide-pane
	hide-spell-tab
	iconify-frame
	minimize-all
	open-compare-window
	open-spell-tab
	resize-to-screen
Shift+Next	scroll-down-half-page
	scroll-down-line
Next	scroll-down-page
	scroll-outline-down-line
	scroll-outline-down-page

	scroll-outline-left
	scroll-outline-right
	scroll-outline-up-line
	scroll-outline-up-page
Shift+Prior	scroll-up-half-page
	scroll-up-line
Prior	scroll-up-page
	simulate-begin-drag
	simulate-end-drag
Ctrl-T	toggle-active-pane
	toggle-invisibles
	toggle-split-direction

8.1.6 Help commands

The following commands print various helpful messages. Apropos commands print longer discussions of specific topics. The help-for-command command prompts for a command name (you can use typing completion to specify the command) and then prints a brief description of that command:

	apropos-autocompletion
	apropos-bindings
	apropos-find-commands
	help
F1	help-for-command
	help-for-python
	mode-help
	print-bindings
	print-commands

8.1.7 Mode commands

These commands put Leo into various kinds of modes.

- The enter-x-mode commands enter modes defined by @mode nodes in leoSettings.leo (or in other .leo files).
- The set-command-state, set-insert-state, set-overwrite-state commands determine how treats unbound plain keys. Leo ignores such keys in command state, inserts them into the body pane in insert state, and overwrites the character at the cursor position in overwrite state.
- Other commands determine whether autocompletion or calltips are in effect.
- When extend mode is effect, basic editing commands that move the cursor also extend the selected text. For example, in extend mode the back-char command works the same as the back-char-extend-selection command.

Here is the full list of mode-related commands:

	auto-complete
Ctrl-Space	auto-complete-force
	disable-autocompleter

	disable-calltips
	enable-autocompleter
	enable-calltips
	enter-apropos-mode
	enter-commands-mode
	enter-edit-mode
	enter-emacs-mode
	enter-extract-mode
	enter-file-mode
	enter-gui-mode
	enter-help-mode
	enter-kill-mode
	enter-modes-mode
	enter-move-outline-mode
	enter-outline-mode
	enter-quick-command-mode
	enter-toggle-find-mode
	exit-named-mode
	set-command-state
	set-extend-mode
	set-insert-state
	set-overwrite-state
	set-silent-mode
	show-calltips
	show-calltips-force
Alt-1	toggle-autocompleter
Alt-2	toggle-calltips
	toggle-extend-mode
	toggle-input-state

8.1.8 Outline commands

The following commands invoke Leo's outline commands:

Ctrl-`	clone-node
Alt-Minus	contract-all
	contract-node
Left	contract-or-go-left
	contract-parent
Ctrl-Shift-C	copy-node
	cut-node
	de-hoist
	delete-node
Ctrl-]	demote
	expand-to-level-1
	expand-to-level-2
	expand-to-level-3
	expand-to-level-4

```
                expand-to-level-5
                expand-to-level-6
                expand-to-level-7
                expand-to-level-8
                expand-to-level-9
                expand-all
                expand-and-go-right
                expand-next-level
Alt-}           expand-node
                expand-or-go-right
                expand-prev-level
                go-back
                go-forward
                goto-first-node
                goto-first-sibling
                goto-last-node
                goto-last-sibling
                goto-last-visible
                goto-line
                goto-line-number
                goto-next-changed
                goto-next-clone
                goto-next-marked
                goto-next-node
                goto-next-sibling
                goto-next-visible
                goto-parent
                goto-prev-node
                goto-prev-sibling
                goto-prev-visible
                hoist
                insert-node
Ctrl-M          mark
                mark-changed-items
                mark-subheads
Ctrl+D          move-outline-down
Shift+Down      move-outline-down
Alt+Shift+Down  move-outline-down
Shift+Left      move-outline-left
Alt+Shift+Left  move-outline-left
Shift+Right     move-outline-right
Alt+Shift+Up    move-outline-up
Ctrl+U          move-outline-up
                outline-to-CWEB
                outline-to-noweb
                paste-node
                paste-retaining-clones
Ctrl-[          promote
```

	sort-children
Alt-A	sort-siblings

8.1.9 Miscellaneous commands

Here are various miscellaneous minibuffer commands:

```
about-leo
add-comments
check-all-python-code
check-outline
check-python-code
clear-recent-files
convert-all-blanks
convert-all-tabs
convert-blanks
convert-tabs
execute-script
export-headlines
exit-leo
extract
extract-names
extract-section
flatten-outline
goto-global-line
import-at-file
import-at-root
import-cweb-files
import-derived-file
import-flattened-outline
import-noweb-files
insert-headline-time
new
open-leoDocs-leo
open-leoPlugins-leo
open-leoSettings-leo
open-offline-tutorial
open-online-home
open-online-tutorial
open-outline
open-outline-by-name
open-python-window
open-with
open-with-idle
open-with-word
open-with-wordpad
pretty-print-all-python-code
pretty-print-python-code
```

```
read-at-file-nodes
read-outline-only
reformat-paragraph
revert
save-buffers-kill-leo
save-file
save-file-as
save-file-to
settings
set-colors
set-font
show-colors
show-find-options
show-fonts
show-invisibles
spell-change
spell-change-then-find
spell-find
spell-ignore
toggle-angle-brackets
weave
write-abbrev-file
write-at-file-nodes
write-dirty-at-file-nodes
write-missing-at-file-nodes
write-outline-only
```

8.2 Autocompletion

Leo's autocompletion feature suggests **completions**, text may be valid in a given point, or **context** in source code. For example, suppose the context is:

```
os.path.s
```

That is, suppose the cursor follows os.path.s in the body pane. The valid completions are all the members of Python's os.path module whose names start with `s', namely:

```
samefile
sameopenfile
sep
split
splitdrive
splitext
splitunc
stat
supports_unicode_filenames
sys
```

How Leo displays these completions depends on the setting:

```
@bool use_qcompleter
```

True: Leo shows completions in the QCompleter popup window. False: Leo shows completions in the log pane.

To compute the list of completions, Leo first computes **Leo-specific** completions. These completions assume that c is a commander, g is the leoGlobals object and p is a position. If there are no such completions, Leo computes completions using ctags data. In order to use these additional completions you must create the ctags data as described in a later section.

8.2.1 Starting autocompletion

There are two ways to have Leo show completions. **Manual autocompletion** shows autocompletions whenever you execute the auto-complete-force (ctrl-space) command. **Automatic autocompletion** shows completions whenever you type a period in the body pane.

You can enable autocompletion in two ways:

1. By setting @bool enable_autocompleter_initially = True.
2. By using the toggle-autocompleter (Alt-1) command.

8.2.2 Using the QCompleter

When the @bool use_qcompleter setting is False, Leo shows all completions in a popup window, regardless of how many completions there are. To **accept** a completion, use the up and down arrows to select a completion, then type the return key. To **cancel** completion, type the escape key. As an important shortcut, if the popup window contains only one entry, you may accept a completion by simply typing the return key.

8.2.3 Using the Log pane completer

When the @bool use_qcompleter setting is True, Leo shows completions in in Leo's log pane. When there are more than 20 completions, Leo shows only the characters that start a completions. For example, when completing os.path. the log pane will show:

```
_ 17
a 2
b 1
c 2
d 3
e 4
g 5
i 5
j 1
l 1
n 2
o 1
p 2
```

```
r 2
s 10
```

To see the complete list, type the `!' character. You will see:

```
__all__
__builtins__
__cached__
__doc__
__file__
__name__
__package__
_get_altsep
_get_bothseps
_get_colon
_get_dot
_get_empty
_get_sep
_get_special
_getfileinformation
_getfinalpathname
_getfullpathname
abspath
altsep
basename
commonprefix
curdir
defpath
devnull
dirname
exists
...
```

Typically, however, you would simply type one of the valid prefix characters. For example, typing the letter `a' would create the context os.path.a and the log pane would show:

```
abspath
altsep
```

As you type, Leo enters the longest common prefix of all completions into the body pane. Typing return, escape or ctrl-g (or any other alt or ctrl key) ends completion.

8.2.4 Showing docstrings

Regardless of the setting of @bool use_qcompleter, typing `?' while autocompleting will show the docstring of the present context. For example, if the context is os.path.join, typing `?' will show:

```
Join two or more pathname components, inserting "\" as needed.
If any component is an absolute path, all previous path components
will be discarded.
```

It is not possible at present to copy the docstring from the log pane when using the QCompleter because the QCompleter popup window is a modal dialog.

8.2.5 Autocompleter settings

These are found in leoSettings.leo: @settings-->Autocompleter:

```
**@bool use_qcompleter = True**
```

True: show completions in a QCompleter popup. False: show completions in Leo's Completions tab.

@bool auto_tab_complete = False

True: Automatically extend the completed text to the longest common prefix of all completions.

@bool autocomplete-brackets = False

True: When typing an opening bracket `(`,'[' or `{`, immediately type the corresponding closing bracket. To move past the closing bracket, just type it.

@bool enable_calltips_initially = False

@bool forbid_invalid_completions = False

True: Don't add characters during autocompletion that are not part of any computed completion.

8.2.6 Creating ctags data

This section describes how to create ctags data files describing the desired completions. You must do this when the use_codewise setting is True.

1. Make sure you have exuberant ctags (not just regular ctags) installed. It's an Ubuntu package, so its easy to install if you're using Ubuntu.
2. Execute the following commands from Leo's external/codewise.py module. **Note**: On Windows, you can use codewise.bat to execute these commands. For example:

```
python <path to leo>\external\codewise.py %*
```

1. [Optional] Create a custom ~/.ctags file containing default configuration settings for ctags:

```
   codewise setup

This command will leave the ~/.ctags file unchanged if it
exists.  Otherwise, the ``codewise setup`` command will
create a ~/.ctags file containing the following defaults::

   --exclude=*.html
   --exclude=*.css

http://ctags.sourceforge.net/ctags.html#FILES for more
details about the .ctags file.
```

2. [Optional] Delete the existing ctags database in ~/.codewise.db:

```
codewise init
```

3. Add ctags data to the existing ctags database:

```
codewise parse <path to directory>
```

   You can add data from multiple sources by running the codewise parse command on multiple directories.

8.3 Calltips

Calltips appear after you type an open parenthesis in code. Calltips shows the expected arguments to a function or method. Calltips work for any Python function or method, including Python's global functions. Typing Return or Control-g (keyboard-quit) exits calltips.

Examples:

1. `g.toUnicode(` gives `g.toUnicode(s, encoding, reportErrors=False'
2. `c.widgetWantsFocusNow' gives `c.widgetWantsFocusNow(w'
3. `reduce(` gives `reduce(function, sequence[, initial]) -> value'

The calltips appear directly in the text and the argument list is highlighted so you can just type to replace it. The calltips appear also in the status line for reference after you have started to replace the args.

Options

Both autocompletion and calltips are initially enabled or disabled by the enable_autocompleter and enable_calltips settings in leoSettings.leo. You may enable or disable these features at any time with these commands: enable-auto-completer-command, enable-calltips-command, disable-auto-completer-command and disable-calltips-command.

8.4 File commands

8.4.1 Loading, Saving and Reverting Files

The new (Ctrl-N) command creates a new Leo main window. The open-outline (Ctrl-O) command opens an existing Leo file and shows it in a main window. The close-window (Ctrl-F4) command closes the selected Leo window, giving you an opportunity to save your work if you haven't yet done so.

The save-file (Ctrl-S), save-file-as and save-file-to commands save the Leo window to a file. The save-files-as command changes the name of the outline being edited; the save-file-to command does not. The save-file-as-zipped command is the same as the save-file-as command except that the resulting .leo file is compressed with Python's zipfile module. Similarly, the save-file-as-unzipped command is the same as the save-as command except that the resulting .leo file is not compressed. The save-file, save-file-as and save-file-to commands compress the file if it was originally compressed. **Note**: Leo writes files with .leo extension, regardless of whether the file is zipped or not. Zipped .leo files contain a single archive, whose name is the same as the .leo file itself. Outside of Leo you can change the extension to .leo.zip and use stuffit or other program to expand the .leo file contained within. The revert command reloads a file, discarding any changes made to the file since it was last saved.

The Recent Files menu shows a list of recently opened files. Choosing an item in this submenu opens the selected file or brings it to the front. The clear-recent-files command deletes all entries in the Recent Files submenu except the most recent file. The files themselves are not affected, just the menu entries.

The following commands are located in the File:Read/Write menu...

The read-outline-only command reads an outline using only the .leo file, not any files derived from @file nodes. This command is useful for reverting a project to a previously saved state. The read-at-file-nodes command updates all @file nodes in an outline. This ensures that the state of an outline matches all files derived from @file nodes. The write-outline-only command saves an outline without writing any @file trees. Useful for inserting an @file node into an outline without modifying a external file with the same name. The write-at-file-nodes command forces an update of all @file trees. The write-dirty-at-file-nodes command writes all @file trees that have been changed.

8.4.2 Communicating with external editors

The open-with command allows you to communicate with external editor. When you select this command Leo creates a temporary file and invokes an external program. Leo periodically checks whether this temporary file has changed; Leo changes the corresponding node in the outline if so. You must create the entries using an @openwith in myLeoSettings.leo. See the documentation in leoSettings.leo.

8.4.3 Importing Files into Leo Outlines

The import-file command imports a file in various ways depending on the contents of the file. For plain files, the command creates an @file node. If the file looks like an external file written by Leo, the import command will recreate the outline structure based on the sentinels in the file. This command can also read files written in MORE outline format.

8.4.4 Exporting Files from Leo Outlines

The outline-to-cweb command creates a CWEB file from the selected outline. The outline-to-noweb command creates a noweb file from the selected outline. The flatten-outline command creates a text file in MORE format from the selected outline. The remove-sentinels command removes all sentinel lines from a file derived from an @file node. The weave command formats the selected text and writes it to a file.

8.4.5 Quitting Leo

The exit-leo (Ctrl-Q or Alt-F4) command causes Leo to exit. You may also exit Leo by closing the main window. You will be prompted to save any file that has been altered but not saved.

8.5 Edit commands

8.5.1 Undoing changes

Leo supports unlimited undo and redo with the undo (Ctrl-Z) and redo (Ctrl-Shift-Z) commands. Think of actions that may be undone or redone as a string of beads. A ``bead pointer" points to the present bead. Performing an operation creates a new bead after the present bead and removes all following beads. Undoing an operation moves the bead pointer backwards; redoing an operation moves the bead pointer forwards. The undo command is disabled when the bead pointer moves in front of the first bead; the redo command is disabled when the bead pointer points to the last bead.

The @string undo_granularity setting controls the granularity of undo. There are four possible values:

node Starts a new undo unit when typing moves to a new node.

line (default) Starts a new undo unit when typing moves to new line.

word Starts a new undo unit when typing starts a new word.

char (not recommended) Starts a new undo unit for each character typed. This wastes lots of computer memory.

8.5.2 Cutting, pasting and selecting text

Leo supports the standard editing commands: cut-text (Ctrl-X), copy-text (Ctrl-C) and paste-text (Ctrl-V), and select-all (Ctrl-A) commands. These commands work with either headline or body text.

8.5.3 Indenting body text

The indent-region (Ctrl-Tab) and unindent-region (Tab) commands shift selected lines in the body text left or right one tab position. These commands shift the entire line if any characters in that line are selected. If no text is selected, the Tab character insert a hard or soft tab depending on the value of the @tabwidth directive in effect.

8.5.4 Adding and deleting comments in body text

The add-comments (Ctrl-)) command puts comments around a block of code. This command uses single-line comments if possible. The delete-comments (Ctrl-() command deletes the comments.

8.5.5 Creating nodes from body text

The extract (Ctrl-Shift-D) command creates a new node whose headline is the first line of selected body text and whose body is all other lines of selected text. Previously selected text is deleted from the original body text. The extract-names (Ctrl-Shift-Command) command creates one or more child nodes, one for each section name in the selected body text. The headline of each created node is the section name.

8.5.6 Converting leading blanks and tabs in body text

The convert-tabs command converts leading tabs to blanks in a single node. The convert-blanks command converts blanks to tabs in a single node. The convert-all-tabs command converts leading tabs to blanks throughout the selected tree. The convert-all-blanks command converts leading blanks to tabs throughout the selected tree. All these commands convert between tabs and blanks using the @tabwidth setting presently in effect.

8.5.7 Executing Python scripts in body text

The execute-script (Ctrl-B) command executes body text as a Python script. Leo execute the selected text, or the entire body text if no text is selected. The Execute Script command pre-defines the values c, g and p as follows:

- c is the commander of the outline containing the script.
- g is the leoGlobals modules.
- p is c.p, that is, c.currentPosition().

Important: Body text may contain Leo directives and section references. You can use all of Leo's features to organize scripts that you execute interactively. Section definitions must appear in the node containing the script or in descendant nodes.

Leo preprocesses all scripts by simulating the writing of a external file to a string. The execute-script command sets app.scriptDict["script1"] to the value of the script before preprocessing, and sets app.scriptDict["script2"] to the value of the script after preprocessing. Scripts may examine and change app.scriptDict as they please.

8.5.8 Finding and changing text

Overview

Leo supports a wide array of commands for searching and replacing text. The typical way to find text is with the search-with-present-options (Ctrl-F). Focus moves to the minibuffer. Type the search pattern, followed by a <Return>. To search and replace, type <Ctrl-F>, followed by the search pattern, followed by replace-string (Ctrl-Shift-R) command, followed by the replacement pattern, and finally a <Return> to start the search.

The following sections discuss all of Leo's find and change commands. **Important**: The radio buttons in the Find tab (Entire Outline, Suboutline Only and Node only) control how much of the outline is affected by Leo's find and change commands.

Basic searches

The search-with-present-options (Ctrl-F) command prompts for a search string. Typing the <Return> key puts the search string in the Find tab and executes a search based on all the settings in the Find tab. This is a recommended default search command. The find-next (F3) command continues a search started with search-with-present-options. The find-previous (F2) commands searches backwards using the present search options.

To search and replace, type <Ctrl-F>, followed by the search pattern, followed by the replace-string (Ctrl-Shift-R) command, followed by the replacement pattern, and finally a <Return> to start the search.

find-all, clone-find-all & clone-find-all-flattened

The find-all command prints all matches in the log pane. The clone-find-all command searches the outline and creates a new root node called Found: *<your search pattern>*. This node contains clones of the found nodes. The clone-find-all-flattened commands includes all found nodes, even if they are also children of previously found nodes.

Change commands

The replace-string (Ctrl-Shift-R) command prompts for a search string. Type <Return> to end the search string. The command will then prompt for the replacement string. Typing a second <Return> key will place both strings in the Find tab and executes a **find** command, that is, search-with-present-options.

The change (Ctrl-=) command replaces the selected text with the `change' text in the Find tab. The change-then-find (Ctrl--) command replaces the selected text with the `change' text in the Find tab, then executes the find command again. These commands can simulate any kind of query-replace command. The change-all changes all occurrences of the `find' text with the `change' text.

Incremental find commands

Incremental find commands move through the text as you type individual characters. Typing <BackSpace> backtracks the search. To repeat an incremental search, type the shortcut for that command again. Here are Leo's incremental find commands:

```
Alt+R isearch-backward
    isearch-backward-regexp
Alt+S isearch-forward
    isearch-forward-regexp
    isearch-with-present-options
```

Commands that set find options

Several commands toggle the checkboxes and radio buttons in the Find tab, and thus affect the operation of the search-with-present-options command. You may bind these commands to keys or toggle these options in a mode. These commands toggle checkboxes:

```
Alt+Ctrl+I  toggle-find-ignore-case-option
Alt+Ctrl+B  toggle-find-in-body-option
Alt+Ctrl+H  toggle-find-in-headline-option
Alt+Ctrl+C  toggle-find-mark-changes-option
Alt+Ctrl+F  toggle-find-mark-finds-option
Alt+Ctrl+X  toggle-find-regex-option
Alt+Ctrl+W  toggle-find-word-option
Alt+Ctrl+A  toggle-find-wrap-around-option
```

These commands set radio buttons:

```
Alt+Ctrl+E  set-find-everywhere
Alt+Ctrl+N  set-find-node-only
Alt+Ctrl+S  set-find-suboutline-only
```

Word search and regex search commands

The following commands set an option in the Find tab, then work exactly like the search-with-present-options command. The search-backward and search-forward commands set the `Whole Word' checkbox to False. The word-search-backward and word-search-forward set the `Whole Word' checkbox to True. The re-search-forward and re-search-backward set the `Regexp' checkbox to True.

Find settings

The following check boxes options appear in the search dialog and control the operations of the find and change commands.

Ignore Case When checked, the Find and Change commands ignore the case of alphabetic characters when determining matches.

Mark Changes When checked, the Change command marks all headlines whose headline or body text are changed by the command.

Mark Matches When checked, the Find and Change commands mark all headlines in which a match is found with the pattern.

Pattern Match When checked, the Find and Change commands treat several characters specially in the find pattern.

- `*' matches any sequence of zero or more characters.
- `.' matches any single character.
- `^' matches a newline at the start of a pattern.
- `$' matches a newline at the end of a pattern.

Examples:

```
"^abc$" matches lines that only contain "abc".
"^a" matches any line starting with "A".
"a$" matches any line ending with "a".
"^*$" matches any line at all.
```

Search Body Text When checked, the Find and Change commands search body text.

Search Headline Text When checked, the Find and Change commands search headline text.

Suboutline Only When checked, the Find and Change commands search only the currently selected headline and its off-spring.

Whole Word When checked, the find pattern must match an entire word. Words consist of an alphabetic character or underscore, followed by zero or more alphabetic characters, numbers or underscores.

Wrap Around When checked, the Find and Change commands continues at the top of the file when the command reaches the bottom of the file. For reverse searches, the find or change command continues at the bottom of the file when the command reaches the top of the file.

8.5.9 Go To Line Number

The goto-global-line (Alt-G) command selects the locations in your outlines corresponding to a line in a external file.

8.5.10 Inserting the date and time

The insert-body-time and insert-headline-time commands insert formatted time and date into body or headline text. You must be editing a headline to be able to insert the time/date into the headline. The body_time_format_string and headline_time_format_string settings specify the format of the inserted text. These settings are the format string passed to time.strftime. For a complete list of the format options see http://www.python.org/doc/current/lib/module-time.html The `` %m/%d/%Y %H:%M:%S" format is used by default, resulting in a time/date format like:

```
1/30/2003 8:31:55
```

8.5.11 Reformatting paragraphs in body text

The reformat-paragraph (Ctrl-Shift-P) command rearranges the words in a text paragraph to fill each line as full as possible, up to the @pagewidth setting. A paragraph is delimited by blank lines, Leo directives, and (of course) start and end of text in a node. The width of the line used by the reformatting operation is governed by @pagewidth and the indentation that would be applied to the node when Leo writes the file.

The command operates on the paragraph containing the insert cursor. If the insert cursor is on a blank line or directive, nothing happens. If the cursor is on a line containing text, then the paragraph containing that text line is reformatted and the insert cursor is moved to the next paragraph.

Note: Hanging indentation is preserved. This is most useful for bulleted or numbered lists, such as:

```
1. This is the first paragraph,
   and it has a hanging indentation.

2. This is the second paragraph,
   and it too has a hanging indentation.
```

8.5.12 Matching brackets and parenthesis

The match-brackets command is enabled if the cursor is next to one of the following characters in the body pane:

> ( ) [ ] { } < >

This command looks for the matching character, searching backwards through the body text if the cursor is next to)] } or > and searching forward through the text otherwise. If the cursor is between two brackets the search is made for the bracket matching the leftmost bracket. If a match is found, the entire range of characters delimited by the brackets is highlighted and the cursor is placed just to the left of the matching characters. Thus, executing this command twice highlights the range of matched characters without changing the cursor.

8.5.13 Indenting body text automatically

Leo auto indents unless @nocolor is in effect. Typing a newline automatically inserts the same leading whitespace present on the previous line.

If Python is the present language, Leo inserts an additional tab if the previous line ends with a colon. When the smart_auto_indent setting is True, Leo uses Emacs-style auto-indentation instead. This style of auto-indent aligns newly created lines with unmatched ([or { brackets in the previous line.

8.5.14 Creating and destroying multiple body editors

The add-editor command adds a new editor in the body pane and gives it the body editor focus. The delete-editor command deletes the editor with body editor focus. The cycle-editor-focus command cycles body editor focus between editors in the body text. The editor that has focus shows the content of the selected outline node; the other body editors continue to show the node contents they last had when they had the body editor focus.

8.5.15 Opening URL's

The open-url (Ctrl-F3 or Ctrl-Left-Click) command looks for a headline in either the headline or body text of the node. If the headline or first body line of the node looks like an URL, the open-url command attempts to open that URL using either os.startfile or a web browser.

Leo checks that the URL is valid before doing so. A valid URL is:

- 3 or more lowercase alphas
- followed by one :
- followed by one or more of:
- $%&'()*+,-./0-9:=?@A-Z_a-z{}~
- followed by one of: $%&'()*+/0-9:=?@A-Z_a-z}~

That is, a comma, hyphen and open curly brace may not be the last character) URL's should contain no spaces: use %20 to indicate spaces. You may use any type of URL that your browser supports: http, mailto, ftp, file, etc.

8.5.16 Using chapters

Chapters are regions of a Leo outline whose root is an @chapter node. They are available in an outline if the @bool usechapters option is True. @chapter nodes may appear anywhere in an outline, but the create-chapter command (see below) creates @chapter nodes as children of the first @chapters (note the s) node in the outline.

One selects a chapter with the select-chapter command, after which Leo shows only the nodes in the selected chapter; in this respect, chapters are like hoists. The main chapter represents the entire outline and can not be deleted by name. When chapters are in effect, Leo creates an @chapters node for the use of create-chapter.

Associated settings:

- The @bool use_chapters setting determines whether chapters are enabled.
- The @bool use_chapter_tabs setting determines whether the chapters pop-up menu appears in the icon area. Choosing a chapter name from this list selects a chapter.

When chapters are enabled, the Cmds->Chapters menu shows all available chapter commands:

- The chapter-create command creates an @chapter node and populates it with a single node.

- The chapter-remove command deletes the currently selected chapter.
- The chapter-select command prompts for a chapter name and makes only the nodes of the selected chapter visible.
- The chapter-move-node-to, chapter-clone-node-to and chapter-copy-node-to commands prompt for a chapter name and add the currently selected node (and its descendants) to another chapter.

8.6 Outline commands

8.6.1 Creating and destroying nodes

The insert-node (Ctrl-I or Insert) command inserts a new node into the outline. When invoked, (from any pane), it inserts a new node below the presently selected node, and at the same level as that node, or at the child level if it has a visible child. The delete-node command deletes a node and all its children. To retain the children, just promote all the children before you do the delete.

8.6.2 Expanding & contracting nodes

You can expand or contract a node by clicking in the tree view icon to the left of the headline. The icon in the Qt gui matches the native OS's tree view icon, i.e. for Mac's, a triangle pointing right or down; on Windows, a square containing a plus or minus. Expanding a node shows its immediate children; contracting a node hides all its children.

The expand-node and contract-node commands also expand and contract nodes. For more convenient navigation, there are expand-and-go-right (Alt-Right) and contract-or-go-up (Alt-Left) commands.

The expand-all command expands every node in the outline. contract-all (Alt-hyphen) contracts every node in the outline. In all but the smallest outlines, expand-all is rarely used, and it does not have a default key binding.

8.6.3 Cutting, pasting and deleting nodes

The cut-node (Ctrl-Shift-X) paste-node (Ctrl-Shift-V), copy-node (Ctrl-Shift-C) and delete-node commands work on nodes rather than text. The cut-node and copy-node commands copy a text representation of the outline to the clipboard. This representation is the same as Leo's .leo file format with some information deleted. You may copy this text representation into a body pane (or into any other text editor) using Edit->Paste in the menus, Ctrl-V, or Alt-X paste-text.

Warning: If you want to preserve the ``cloned" attribute of a node, or want to paste the node as a clone of the node you cut or copied, use the past-retaining-clones command, which in the Outline menu is called ``Paste Node as Clone". The paste-node command instead creates a new, distinct version of the node you previously cut or copied, though if there were descendant nodes which were clones of each other, the new version will have parallel, distinct nodes that are also clones of each other (just not of the originals). You may paste a node between .leo files, but there can be no clone relationship across files.

The paste-retaining-clones command is disabled if it would cause a node to become a parent of itself. The Leo outline is thus mathematically a *directed acyclic graph*: clones make it more flexible than a tree, but not a generalized graph.

8.6.4 Navigating through the outline

Leo has many commands that select nodes in the outline. These commands can be found in the Outline:Go To menu.

As described in the tutorial, you can move about the outline by clicking on the headlines or using Alt+arrow keys.

8.6.5 Moving & Reorganizing nodes

The move-outline-up (Ctrl-U or Alt-Shift-Up), move-outline-down (Ctrl-D or Alt-Shift-Down), move-outline-left (Ctrl-L or Alt-Shift-Left), and move-outline-right (Ctrl-R or Alt-Shift-Right) commands move the currently selected node. **Important**: When focus is in the outline pane, you can move nodes without adding the Alt modifier. Shift-Up moves the select node up, etc.

The promote (Ctrl-[) command makes all the children of a node siblings of the node. The demote (Ctrl-]) command makes all following siblings of a node children of the node.

8.6.6 Cloning nodes

A cloned node is a copy of a node that changes when the original changes. One may also think of it as a single node that is hooked into the outline at multiple positions. Because that single node brings along all its descendants, changes are maintained across all the the clones of a node, along with changes to its offspring (children, grandchildren, etc.), i.e., any changes are simultaneously made to the corresponding offspring of all of those clones. A small red arrow in the icon box marks cloned nodes. You can think of the arrow as pointing out that there are other paths to get to this same node. There is no real distinction between the ``original" node and any of its clones. Any headline or body update of a clone headed subtree affects all of its clones simultaneously. A cloned node becomes a regular node whenever deletion of its other clones makes it the only one left. Clones are useful for making alternate views of a program. See Clones and views for full details.

The clone-node (Ctrl-`) command creates a clone as the immediate sibling of a selected node. You have to place it where you want it by either using move commands, or cutting and paste the clone.

8.6.7 Marking nodes

The mark (Ctrl-M) marks a node if it is unmarked, and unmarks the node if it is already marked. The mark-subheads command marks all offspring of the presently selected node. The mark-changed-items command marks all nodes whose headline or body text has been changed since the file was last saved.

Leo's find and change commands mark nodes if the ``Mark Changes" and ``Mark Finds" checkboxes are checked. You can change these checkboxes with the toggle-find-mark-changes-option and toggle-find-mark-finds-option commands.

The goto-next-marked command selects the next marked node.

8.6.8 Dragging nodes

You may drag a node (including all its descendants) from one place to another in an outline. To start a drag, press the main (left) mouse button while the cursor is over the icon for a node. The cursor will change to a hand icon. If you release the mouse button while the hand cursor is above another node, Leo will move the dragged node after that node. If you release the mouse button when the hand cursor is not over a node, Leo will leave the outline pane as it is. Leo scrolls the outline pane as the result of mouse-moved events, so to continue scrolling you must keep moving the mouse.

If the recipient node has children and is expanded, the dropped node will be inserted as the first child of the recipient node, otherwise the dropped node will be inserted after the recipient node.

Holding down Alt before releasing the node will force insertion as a child of the recipient node, even if the recipient node is not expanded.

Holding down Control before releasing the node will cause a clone to be dropped, leaving the original where it was.

8.6.9 Hoisting & De-hoisting nodes

The hoist command redraws the screen so presently selected tree becomes the only visible part of the outline. You may hoist an outline as many times as you wish. The dehoist command undoes the effect of the previous hoist command.

8.6.10 Checking outlines

The check-outline command checks the outline for consistency. Leo automatically check the syntax of Python external files when Leo writes the external file.

The pretty-print-python-code and pretty-print-all-python-code pretty print body text. You can customize this code by overriding the following methods of class prettyPrinter in leoCommands.py:

```
putOperator:      puts whitespace around operators.
putNormalToken:   puts whitespace around everything else.
```

8.6.11 Resizing panes

You can change the relative sizes of the outline and body panes by dragging the splitter bar. The equal-sized-panes command resizes the panes so that each fills half of the main window.

8.7 Window commands

- The Equal Sized Panes command adjusts the sizes of the outline and body panes so that they are the same height.
- The Cascade command cleans up the screen by cascading all Leo windows.
- The Minimize All command minimizes all Leo windows.
- The Toggle Active Pane command toggles keyboard focus between the outline and body panes.
- The Toggle Split Direction command switches between vertical and horizontal orientations of the Leo window. In the vertical orientation, the body pane appears below the pane containing the outline and log panes. In the horizontal orientation, the body pane appears to the left the pane containing the outline and log panes. By default, the ratio of pane outline pane to the body pane is 0.5 in the vertical orientation and 0.3 in the horizontal orientation. These two ratios may be changed using settings.

- The Open Compare Window command opens a dialog that allows you to compare two files, one containing sentinels and one not.

8.8 Help commands

- The About Leo command puts up a dialog box showing the version of Leo.
- The Online Home Page command opens Leo's home page at http://webpages.charter.net/edreamleo/front.html.
- The Open Online Tutorial command opens Joe Orr's excellent ScreenBook tutorial at http://www.evisa.com/e/sbooks/leo/sbframetoc_ie.htm.
- The Open Offline Tutorial command opens the file sbooks.chm if it exists. Otherwise, you will be asked whether you want to download it from Leo's SourceForge web site. If you say yes, the page http://sourceforge.net/project/showfiles.php?group_id=3458 will open. You may then download sbooks.sbm to the folder containing leo.py.
- The Open LeoDocs.leo command opens LeoDocs.leo.
- The Open LeoPlugins.leo command opens LeoPlugins.leo.
- The Open LeoSettings.leo command opens LeoSettings.leo.

第 9 章

Customizing Leo

This chapter discusses how to customize Leo using the plugins and other means. See Specifying settings for a description of how to change Leo's settings.

Contents

- Customizing Leo
  - Specifying settings
    * Configuration directories
    * Search order for settings files
    * Safe rules for local settings
    * Organizer nodes
    * @ignore and @if nodes
    * Simple settings nodes
    * Complex settings nodes
  - Input modes
  - Adding extensible attributes to nodes and .leo files
  - Translating Leo's menus and messages
  - Writing new importers

9.1 Specifying settings

Leo stores options in **@settings trees**, outlines whose headline is @settings. When opening a .leo file, Leo looks for @settings trees not only in the outline being opened but also in various leoSettings.leo files. This scheme allows for the following kinds of settings:

- Per-installation or per-machine settings.

- Per-user settings.
- Per-folder settings.
- Per-file settings.

There are four kinds of settings files:

1. **Default settings files**, named **leoSettings.leo**. Although they can be used in other ways, they typically contain default settings.
2. **Personal settings files**, named **myLeoSettings.leo**. They provide a way of ensuring that your customized settings are not altered when updating Leo from bzr or while installing a new version of Leo. The myLeoSettings.leo acts much like Python's site-customize.py file. myLeoSettings.leo will never be part of any Leo distribution, and it will never exist in Leo's cvs repository. This solution is *much* better than trying to update leoSettings.leo with scripts.
3. **Machine settings files**, named **LeoSettings.leo** (note the capital `L'), and appearing in a unique directory.
4. **Command-line settings files**, specified using Leo's -c command-line option. Any .leo file may be used, provided it has an @settings tree. These files typically provide a common set of settings for files scattered in various places on the file system.

The following sections describe the kinds of nodes in @settings trees.

9.1.1 Configuration directories

Settings files can be found in the following directories:

- **homeDir**, the HOME/.leo directory. HOME is given by Python's HOME environment variable, or by os.expanduser (`~') if no HOME environment variable exists.
- **configDir**, Leo's configuration directory: leo/config.
- **machineDir**, the HOME/.leo/MACHINE directory. MACHINE is given by Python's HOSTNAME environment variable, or by Python's COMPUTERNAME environment variable if there is no HOSTNAME variable, or by the value returned by socket.gethostname() if neither environment variable exists.
- **localDir**, the directory containing the .leo file being loaded.

In addition, Leo's -c command-line option can specify any .leo file anywhere.

9.1.2 Search order for settings files

When reading a .leo file, Leo looks for settings in default settings files first, then settings in personal settings files, and finally settings in local settings files. The exact search order is:

1. Default settings files:
   (a) configDir/leoSettings.leo
   (b) homeDir/leoSettings.leo
   (c) localDir/leoSettings.leo
2. Personal settings files:

   (a) configDir/myLeoSettings.leo
   (b) homeDir/myLeoSettings.leo
   (c) homeDir/<machine-name>LeoSettings.leo (note capitalization)
   (d) localDir/myLeoSettings.leo

3. Local settings files:
   (a) The file specified by the -c command-line option.
   (b) The file being loaded.

Settings that appear later in this list override settings that appear earlier in this list. This happens on a setting-by-setting basis, *not* on a file-by-file basis. In other words, each individual setting overrides only the *corresponding* setting in previously-read files. Reading a setting file does *not* reset all previous settings. Note that the same file might appear several times in the search list. Leo detects such duplicate file names and only loads each settings file once. Leo remembers all the settings in settings files and does not reread those settings when reading another .leo file.

Caution: This search order offers almost too much flexibility. This can be confusing, even for power users. It's important to choose the ``simplest configuration scheme that could possibly work". Something like:

- Use a single leoSettings.leo file for installation-wide defaults.
- Use a single myLeoSettings.leo files for personal defaults.
- Use local settings sparingly.

Important: it is good style to limit settings placed in myLeoSettings.leo to those settings that differ from default settings.

9.1.3 Safe rules for local settings

You should use special care when placing default or personal settings files in **local** directories, that is, directories other than homeDir, configDir or machineDir. In particular, the value of localDir can change when Leo reads additional files. This can result in Leo finding new default and personal settings files. The values of these newly-read settings files will, as always, override any previously-read settings.

Let us say that a setting is **volatile** if it is different from a default setting. Let us say that settings file A.leo **covers** settings file if B.leo if all volatile settings in B.leo occur in A.leo. With these definitions, the **safe rule** for placing settings files in local directories is:

```
Settings files in local directories should
cover all other settings files.
```

Following this rule will ensure that the per-directory defaults specified in the local settings file will take precedence over all previously-read default and personal settings files. Ignore this principle at your peril.

9.1.4 Organizer nodes

Organizer nodes have headlines that do no start with @. Organizer nodes may be inserted freely without changing the meaning of an @setting tree.

9.1.5 @ignore and @if nodes

Leo ignores any subtree of an @settings tree whose headline starts with @ignore.

You can use several other kinds of nodes to cause Leo to ignore parts of an @settings tree:

- @if *expression*

  A node whose headline starts with @if *expression* acts like an organizer node if the expression evaluates to True, otherwise acts like an @ignore node. If the expression is empty the body text should contain a script that will be evaluated (in an empty context).

- @ifplatform *platform-name*

  Same as @if sys.platform == ``platform-name'': except that it isn't necessary to import sys.

- @ifhostname *hostA,!hostB*

  Evaluates to True if and only if: h=g.computeMachineName(); h==hostA and h!=hostB. The "!" version allows matching to every machine name except the given one to allow differing settings on only a few machines.

9.1.6 Simple settings nodes

Simple settings nodes have headlines of the form:

```
@<type> name = val
```

set the value of name to val, with the indicated type.

<type> may be one of the following:

<type>	Valid values
@bool	True, False, 0, 1
@color	A Tk color name or value, such as `red' or `xf2fddff' (without the quotes)
@directory	A path to a directory
@float	A floating point number of the form nn.ff.
@int	An integer
@ints[list]	An integer (must be one of the ints in the list). Example: @ints meaningOfLife[0,42,666]=42
@keys[name]	Gives a name to a set of bindings for the Check Bindings script in leoSettings.leo.
@path	A path to a directory or file
@ratio	A floating point number between 0.0 and 1.0, inclusive.
@string	A string
@strings[list]	A string (must be one of the strings in the list). Example: @strings tk_relief['flat','groove','raised']='groove'

Note: For a list of Tk color specifiers see:

- http://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm
- http://www.tcl.tk/man/tcl8.4/TkLib/GetColor.htm

Important: you can use the show-colors minibuffer command to guide you in making these settings.

9.1.7 Complex settings nodes

Complex settings nodes have headlines of the form:

```
@<type> description
```

The type may be one of the following:

<type>	Valid values
@buttons	Child @button nodes create global buttons
@commands	Child @command nodes create global buttons
@data	Body text contains a list of strings, one per line.
@enabled-plugins	Body text contains a list of enabled plugins
@font	Body text contains a font description
@menus	Child @menu and @item nodes create menus and menu items.
@menuat	Child @menu and @item nodes modify menu tree create by @menus.
@mode [name]	Body text contains a list of shortcut specifiers.
@recentfiles	Body text contains a list of file paths.
@shortcuts	Body text contains a list of shortcut specifies.

Complex nodes specify settings in their body text. See the following sections for details.

@button

An @buttons tree in a settings file defines global buttons that are created in the icon area of all .leo files. All @button nodes in the @commands tree create global buttons. All @button nodes outside the commands tree create buttons local to the settings file.

@commands

An @commands tree in a settings file defines global commands. All @command nodes in the @commands tree create global commands. All @command nodes outside the commands tree create commands local to the settings file.

@data

The body text contains a list of strings, one per line. Lines starting with `#' are ignored.

@enabled-plugins

The body text of the @enabled plugins node contains a list of enabled plugins, one per line. Comment lines starting with `#' are ignored. Leo loads plugins in the order they appear. **Important**: Leo handles @enabled-plugins nodes a differently from other kinds of settings. To avoid confusion, **please read the following carefully**.

As always, Leo looks for @enabled-plugins nodes in settings files in the order specified by Search order for settings files. Leo will enable all plugins found in the @enabled-plugins node it finds *last* in the search order. Leo does *not* enable plugins found in any other @enabled-plugins node. In particular, **you can not specify a list of default plugins by placing that list**

in a settings file that appears early in the search list. Instead, the last @enabled-plugins node found in the search list specifies all and *only* the plugins that will be enabled.

Let us distinguish two different situations. First, what Leo does when loading a file, say x.leo. Second, what Leo does when loading a second file, say y.leo, *from x.leo*. When loading the first .leo file, Leo enables plugins from the @enabled-plugins node it finds *last* in the search order. But after plugins have *already* been loaded and enabled, there is no way to disable previously loaded-and-enabled plugins. But local settings files can enable additional plugins.

To avoid confusion, I highly recommend following another kind of safe rule. We say that an @enabled-plugin node in file A.leo **covers** an @enabled-plugin node in file B.leo if all plugins specified in B's @enabled-plugin node appear A's @enabled-plugin node. The safe rule for plugins is:

```
@enabled-plugin nodes in settings files in local directories
should cover @enabled-plugins nodes in all other settings files.
```

@font

The body text contains a list of settings for a font. For example:

```
body_text_font_family = Courier New
body_text_font_size = None
body_text_font_slant = None
body_text_font_weight = None
```

Important: you can use the show-fonts minibuffer command to guide you in making these settings.

@menuat

The form of this node is:

```
@menuat *<path>* *<action>* *[clipboard]*
```

The @menuat setting has 2-3 parameters in its head text, its children are @menu and @item nodes as for the @menu setting. @menuat modifies the menu tree created by @menus. It is intended to be used in myLeoSettings.leo to modify the menu tree created in leoSettings.leo. This allows you to change the menus without having to re-create the entire menu tree from leoSettings.leo, and ensures you don't miss out when new things are added in the @menus in leoSettings.leo, as you would if you replaced the @menus in leoSettings.leo with one in myLeoSettings.leo. @menuat should occur in a @settings tree, but not as a descendant of a @menus tree. There is an example of the use of the @menuat setting in the file .../leo/core/test/menuAtTest.leo.

The **path** argument specifies a target location in the menu tree as defined by @menus and modified by earlier @menuat settings. The path takes the form /entry1/entry2/entry3 where each entry is the name of a menu or item with all text except a-z and 0-9 removed. Upper case letters are converted to lower case. So for example to use the *Outline->Move->Move Down* menu item as a target, you would specify a path as */outline/move/movedown*.

The **action** argument specifies what the menu item does. There are 5 available actions:

- **before**: The supplied items and sub menus will be inserted immediately before the target menu or item.
- **after**: The supplied items and sub menus will be inserted immediately after the target menu or item.

- **append**: The supplied items and sub menus will be appended at the end of the menu containing the target menu or item.
- **cut**: The target menu or item will be removed from the menu tree and saved to an internal clipboard. This can be used for deleting menus or items. Descendants of the @menuat setting are ignored.
- **copy**: The target menu or item will be copied and saved to an internal clipboard. Descendants of the @menuat setting are ignored.

The optional **clipboard** argument modifies the action of the before, after, and append actions. By default these actions insert the menus and items supplied as descendants of the @menuat setting. If you specify ``clipboard'' (without the quotes) as the source, the contents of the clipboard from a previous cut or copy action will be used instead. This parameter is optional and can be left blank.

@menus

Leo creates its menus from the @menu, @item and @popup nodes in the @menus tree. Within @menus trees, @menu nodes create menus and @item nodes create menu items.

The menu name always follows @menu. If the menu name is `Plugins', Leo will create the Plugins menu and populate the menu by calling the `create-optional-menus' hook. This creates the Plugins menu as usual. Nested @menu nodes define submenus.

The command name follows @item. If the body text of an @item node exists, this body text is the menu name. Otherwise, the menu name is the command name. However, if the command name starts with a `*', hyphens are removed from the menu name. Menu names and command names may contain a single ampersand (&). If present, the following character is underlined in the name. If the command name in an @item node is just a hyphen (-), the item represents a menu separator.

@popup <*widget-name*> creates a popup menu for use by the rClick plugin. The children of this node should be @menu and @item nodes, used as with @menus.

@mode

Leo now allows you to specify input modes. You enter mode x with the enter-x-mode command. The purpose of a mode is to create different bindings for keys within a mode. Often plain keys are useful in input modes.

You can specify modes with @mode nodes in leoSettings.leo. @mode nodes work just like @shortcuts nodes, but in addition they have the side effect of creating the enter-<mode name>-mode command.

The form of this node is:

```
@mode *<mode name>*
```

The body text contains a list of shortcut specifiers. @mode nodes work just like @shortcuts nodes, but in addition they have the side effect of creating the enter-<mode name>-mode command.

Notes:

- You can exit any mode using the keyboard-quit (Control-g) command. This is the **only** binding that is automatically created in each mode. All other bindings must be specified in the @mode node. In particular, the bindings specified in @shortcuts nodes are **not** in effect in mode (again, except for the keyboard-quit binding).

- Leo supports something akin to tab completion within modes: if you type a key that isn't bound in a mode a `Mode' tab will appear in the log pane. This tab shows all the keys that you can type and the commands to which they are bound. The mode-help command does the same thing.
- @shortcuts nodes specify the bindings for what might be called the `top-level' mode. These are the bindings in effect when no internal state is present, for example, just after executing the keyboard-quit command.
- The top_level_unbound_key_action setting determines what happens to unbound keys in the top-level mode. Leo ignores unbound keys in all other modes. The possibilities are `insert', `replace' and `ignore'.
- The set-insert-mode, set-overwrite-mode and set-ignore-mode commands alter what happens to unbound keys in the top-level mode.
- If the @mode headline contains ::, everything following the :: is the mode prompt. For example:

```
@mode abc :: xyz
```

Creates the enter-abc-mode command, but the prompt for the command is xyz.

With all these options it should be possible to emulate the keyboard behavior of any other editor.

@recentfiles

The body text contains a list of paths of recently opened files, one path per line. Leo writes the list of recent files to .leoRecentFiles.txt in Leo's config directory, again one file per line.

@shortcuts

The body text contains a list of shortcut specifiers.

9.2 Input modes

Leo now allows you to specify input modes. You enter mode x with the enter-x-mode command. The purpose of a mode is to create different bindings for keys within a mode. Often plain keys are useful in input modes.

You can specify modes with @mode nodes in leoSettings.leo. @mode nodes work just like @shortcuts nodes, but in addition they have the side effect of creating the enter-<mode name>-mode command.

Notes:

- You can exit any mode using the keyboard-quit (Control-g) command. This is the **only** binding that is automatically created in each mode. All other bindings must be specified in the @mode node. In particular, the bindings specified in @shortcuts nodes are **not** in effect in mode (again, except for the keyboard-quit binding).
- Leo supports something akin to tab completion within modes: if you type a key that isn't bound in a mode a `Mode' tab will appear in the log pane. This tab shows all the keys that you can type and the commands to which they are bound. The mode-help command does the same thing.
- @shortcuts nodes specify the bindings for what might be called the `top-level' mode. These are the bindings in effect when no internal state is present, for example, just after executing the keyboard-quit command.

- The top_level_unbound_key_action setting determines what happens to unbound keys in the top-level mode. Leo ignores unbound keys in all other modes. The possibilities are `insert', `replace' and `ignore'.
- The set-insert-mode, set-overwrite-mode and set-ignore-mode commands alter what happens to unbound keys in the top-level mode.
- If the @mode headline contains ::, everything following the :: is the mode prompt. For example:

```
@mode abc :: xyz
```

Creates the enter-abc-mode command, but the prompt for the command is xyz.

With all these options it should be possible to emulate the keyboard behavior of any other editor.

9.3 Adding extensible attributes to nodes and .leo files

Leo's .leo file format is extensible. The basis for extending .leo files are the v.unknownAttributes ivars of vnodes, uA's for short. Leo translates between uA's and xml attributes in the corresponding <v> elements in .leo files. Plugins may also use v.tempAttributes ivars to hold temporary information that will *not* be written to the .leo file. These two ivars are called **attribute ivars**.

Attribute ivars must be Python dictionaries, whose keys are names of plugins and whose values are *other* dictionaries, called **inner dictionaries**, for exclusive use of each plugin.

The v.u Python property allows plugins to get and set v.unknownAttributes easily:

```
d = v.u # gets uA (the outer dict) for v
v.u = d # sets uA (the outer dict) for v
```

For example:

```
plugin_name = 'xyzzy'
d = v.u # Get the outer dict.
inner_d = d.get(plugin_name,{}) # Get the inner dict.
inner_d ['duration']= 5
inner_d ['notes'] "This is a note."
d [plugin_name] = inner_d
v.u = d
```

No corresponding Python properties exist for v.tempAttributes, so the corresponding example would be:

```
plugin_name = 'xyzzy'
# Get the outer dict.
if hasattr(p.v,'tempAttributes'): d = p.v.tempAttributes
else: d = {}
inner_d = d.get(plugin_name,{}) # Get the inner dict.
inner_d ['duration'] = 5
inner_d ['notes'] = "This is a note."
d [plugin_name] = inner_d
p.v.tempAttributes = d
```

Important: All members of inner dictionaries should be picklable: Leo uses Python's Pickle module to encode all values in these dictionaries. Leo will discard any attributes that can not be pickled. This should not be a major problem to plugins. For example, instead of putting a tnode into these dictionaries, a plugin could put the tnode's gnx (a string) in the dictionary.

Note: Leo does *not* pickle members of inner dictionaries whose name (key) starts with str_. The values of such members should be a Python string. This convention allows strings to appear in .leo files in a more readable format.

Here is how Leo associates uA's with <v> elements in .leo files:

- **Native xml attributes** are the attributes of <v> elements that are known (treated specially) by Leo's read/write code. The native attributes of <v> elements are a, t, vtag, tnodeList, marks, expanded and descendentTnodeUnknownAttributes. All other attributes of <v> and <t> elements are **foreign xml attributes**.
- When reading a .leo file, Leo will create v.unknownAttributes ivars for any vnode whose corresponding <v> or <t> element contains a foreign xml attribute.
- When writing a file, Leo will write foreign xml attributes in <v> elements if the corresponding vnode contains an unknownAttributes ivar.
- Leo performs the usual xml escapes on these strings when reading or writing the unknownAttributes ivars.

9.4 Translating Leo's menus and messages

It is easy to translate Leo's menu strings: simply create an @menus tree in leoSettings.leo or myLeoSettings.leo that contains the translated menu names.

New in Leo 4.4.8: Leo now contains support for translating messages sent to Leo's log:

- Rather than using an `_' function to denote strings to be translated, Leo's g.es and g.es_print functions translate ``odd'' (first, third, fifth) arguments, leaving ``even'' arguments untranslated. Keyword arguments, color, newline, etc. are never translated.
- All calls to g.es and g.es_print in Leo's core follow this convention.
- g.translateString does the actual translation using Python's gettext module.
- You can use the script in the node ``@button print g.es stats'' in scripts.leo to create catalogs of all scripts that need to be translated. Such catalogs are used by Python's gettext module. (This script was also used to check that the proper arguments to g.es and g.es_print were translated.)

9.5 Writing new importers

This section describes the process of creating an importer for a new language. There are a set of ``importers'' in leoImport.py, all based on the baseScannerClass class. You can define your own importer by creating a subclass. This shouldn't be too difficult: baseScannerClass is supposed to do almost all the work. With luck, your subclass might be very simple, as with class cScanner.

Important As I write this, I realize that I remember very little about the code, but I do remember its general organization and the process of creating a new importer. The following should be all you need to write any importer.

This base class has three main parts:

1. The ``parser" that recognizes where nodes begin and end.
2. The ``code generator" the actually creates the imported nodes.
3. Checking code that ensures that the imported code is equivalent to the original code.

You should never have to change the code generators or the checking code. Confine your attention to the parser.

The parser thinks it is looking for classes, and within classes, method definitions. Your job is to tell the parser how to do this. Let's look at part of the ctor for baseScannerClass for clues:

```
# May be overridden in subclasses.
self.anonymousClasses = [] # For Delphi Pascal interfaces.
self.blockCommentDelim1 = None
self.blockCommentDelim2 = None
self.blockCommentDelim1_2 = None
self.blockCommentDelim2_2 = None
self.blockDelim1 = '{'
self.blockDelim2 = '}'
self.blockDelim2Cruft = [] # Stuff that can follow .blockDelim2.
self.classTags = ['class',] # tags that start a tag.
self.functionTags = []
self.hasClasses = True
self.hasFunctions = True
self.lineCommentDelim = None
self.lineCommentDelim2 = None
self.outerBlockDelim1 = None
self.outerBlockDelim2 = None
self.outerBlockEndsDecls = True
self.sigHeadExtraTokens = [] # Extra tokens valid in head of signature.
self.sigFailTokens = []
    # A list of strings that abort a signature when seen in a tail.
    # For example, ';' and '=' in C.
self.strict = False # True if leading whitespace is very significant.
```

Naturally, this looks like gibberish at first. I do *not* remember what all these things do in detail, although obviously the names mean something. What I *do* remember is that these ivars control the operation of the startsFunction and startsClass methods and their helpers (especially startsHelper) and the methods that call them, scan and scanHelper. Most of these methods have a trace var that will enable tracing during importing.

So the strategy is simple: study startsHelper in detail, set the ivars above to make startsHelper do what you want, and trace until things work as you want.

There is one more detail. Sometimes the ivars above are not sufficient to get the job done. In that case, subclasses will override various methods of the parser, but *not* the code generator. If indentation is important, you will want to look at the Python importer. Notice that it overrides skipCodeBlock, called by startsHelper.

That's about it. It would be pointless to give you more details, because those details would lead you *away* from the process you need to follow. Having said that, feel free to ask further questions. I'll be glad to answer them.

第 **10** 章

Controlling Syntax Coloring

This chapter discusses the settings to control Leo's syntax colorer. This chapter also discusses how to extend Leo's colorizer by creating xml language descriptions files and corresponding Python files. **Important**: this material is for those who want to support Leo's colorizing code. To use Leo's colorizers you only need to know about syntax-coloring settings.

Contents

- Controlling Syntax Coloring
  - Syntax coloring settings
  - Files
  - The colorizer's inner loop
  - Format of colorizer control files
    * Ruleset names
    * x.properties
    * Attribute dictionaries and x.attributesDictDict
    * Keyword dictionaries and x.keywordsDictDict
    * Rules, rules dictionaries and x.rulesDictDict
    * x.importDict and imported versus delegated rulesets
  - Rule methods
    * Arguments to rule methods
    * match_eol_span
    * match_eol_span_regexp
    * match_keywords
    * match_mark_following
    * match_mark_previous
    * match_seq
    * match_seq_regexp
    * match_span
    * match_span_regexp
    * match_terminate

10.1 Syntax coloring settings

This section discusses only those settings that affect syntax coloring. See Customizing Leo for a general discussion of Leo's settings.

Both the old colorizer (in Leo's core) and the new colorizer (the threading_colorizer and qtGui plugins) now support @color and @font settings for colorizing options. The settings for the old colorizer are:

```
comment_font, cweb_section_name_font, directive_font,
doc_part_font, keyword_font, leo_keyword_font, section_name_font,
section_name_brackets_font, string_font, undefined_section_name_font,
latexBackground_font, and latex_background_font.
```

The settings for the new colorizer are all of the above (except keyword_font) plus the following:

```
comment1_font, comment2_font, comment3_font, comment4_font, function_font,
keyword1_font, keyword2_font, keyword3_font, keyword4_font, label_font,
literal1_font, literal2_font, literal3_font, literal4_font, markup_font,
null_font, and operator_font.
```

To specify a color, say for comment1, for *all* languages, create an @color node:

```
@color comment1_color = blue
```

To specify a color for a **particular** language, say Python, prepend the setting name with the language name. For example:

```
@color python_comment1_color = pink
```

To specify a font, say for keyword_font, to be used as the default font for **all** languages, put the following in the body text of an @font node in leoSettings.leo:

```
# keyword_font_family = None
keyword_font_size = 16
keyword_font_slant = roman
   # roman, italic
keyword_font_weight = bold
   # normal, bold
```

Comments are allowed and undefined settings are set to reasonable defaults. At present, comments can not follow a setting: comments must start a line.

You can specify per-language settings by preceding the settings names by a prefix x. Such settings affect only colorizing for language x (i.e., all the modes in modes/x.py when using the new colorizer). For example, to specify a font for php (only), put the following in the body text of an @font node in leoSettings.leo:

```
# php_keyword_font_family = None
php_keyword_font_size = 16
php_keyword_font_slant = roman
   # roman, italic
php_keyword_font_weight = bold
   # normal, bold
```

10.2 Files

The jEdit editor drives its syntax colorer using xml **language description files.** Rather than using the xml language description files directly, Leo uses Python **colorer control files**, created automatically from the xml files by a script called jEdit2Py. All these files reside in the leo/modes directory.

These Python files contain all the information in the jEdit's xml files, so we can (loosely) speak of modes, rulesets, rules, properties and attributes in the Python colorer control files. Later sections of this documentation will make this loose correspondence exact.

jEdit's documentation contain a complete description of these xml files. Each xml file describes one **colorizing mode**. A mode consists of one or more **rulesets**, and each ruleset consists of a list of **colorizing rules**. In addition, modes, rulesets and rules may have associated **properties** and **attributes**. Various rules may specify that the colorizer uses another ruleset (either in the same mode or another mode).

Important: jEdit's xml language description files contain no explicit <RULE> elements Rules are simply sub-elements of an enclosing <RULES> element. The element indicates the kind of rule that is specified, for example, <SPAN>, <SEQ>, etc. By the term **rule element** we shall mean any sub-element of the <RULES> element.

Important: throughout this documentation, **x.py** will refer to the Python colorer for language x, and **x.xml** will refer to the corresponding xml language-description file.

Using Python colorer control files has the following advantages:

- Running jEdit2Py need only be done when x.xml changes, and the speed of the xml parser in jEdit2Py does not affect the speed of Leo's colorizer in any way. Moreover, the jEdit2Py script can contain debugging traces and checks.
- Colorer control files are valid .py files, so all of Python's import optimizations work as usual. In particular, all the data in colorer control files is immediately accessible to Leo's colorer.
- Colorer control files are easier for humans to understand and modify than the equivalent xml file. Furthermore, it is easy to insert debugging information into Python colorer control files.
- It is easy to modify the Python colorer control files `by hand' without changing the corresponding xml file. In particular, it would be easy to define entirely new kinds of pattern-matching rules in Python merely by creating functions in a colorer control file.

10.3 The colorizer's inner loop

When Leo's syntax colorer sees the `@language x' directive, it will import x.py from Leo's modes folder. The colorer can then access any module-level object obj in x.py as x.obj.

Colorizer control files contain **rules functions** corresponding to rule elements in x.xml. The colorizer can call these functions as if they were members of the colorizer class by passing `self' as the first argument of these functions. I call these rules *functions* to distinguish them from the corresponding **rules methods** which are actual methods of the colorizer class. Rules *functions* merely call corresponding rules *methods*. Indeed, rules functions are simply a way of binding values to keyword arguments of rules methods. These keywords arguments correspond to the xml attributes of rule elements in x.xml.

The colorizer calls rules functions until one matches, at which point a range of text gets colored and the process repeats. The inner loop of the colorizer is this code:

```
for f in self.rulesDict.get(s[i],[]):
    n = f(self,s,i)
    if n > 0:
        i += n ; break
    else: i += 1
```

- rulesDict is a dictionary whose keys are rulesets and whose values are ruleset dictionaries. Ruleset dictionaries have keys that are single characters and whose values are the list of rules that can start with that character.
- s is the full text to be colorized.
- i is the position within s is to be colorized.

Rules methods (and functions) return n > 0 if they match, and n == 0 if they fail.

10.4 Format of colorizer control files

The following sections describe the top-level data in x.py.

10.4.1 Ruleset names

A **ruleset name** is a Python string having the form `x_setname', where setname is the value of the SET attribute of the <RULES> element in x.xml. For example, the ruleset name of the ruleset whose SET attribute is JAVASCRIPT in php.xml is `php_JAVASCRIPT'. **Important**: by convention, the ruleset name of the default <RULES> element is `x_main'; note that default <RULES> element have no SET attributes.

The colorizer uses ruleset names to gain access to all data structures in x.py. To anticipate a bit, ruleset names are keys into two standard dictionaries, x.rulesDict and x.keywordsDictDict, from which the colorizer can get all other information in x.py:

```
# The rules list for the 'JAVASCRIPT' ruleset in php.xml.
rules = x.rulesDict('php_JAVASCRIPT')

# The keywords dict for the 'JAVASCRIPT' ruleset in php.xml.
keywordsDict = x.keywordsDictDict('php_JAVASCRIPT')
```

In fact, ruleset names (and x.rulesDict and x.keywordsDictDict) are the **only** names that the colorizer needs to know in order to access all information in x.py.

10.4.2 x.properties

x.properties is a Python dictionary corresponding to the <PROPS> element in x.xml. Keys are property names; values are strings, namely the contents of <PROPERTY> elements in x.xml. x.properties contains properties for the entire mode. That is, only modes have <PROPS> elements. For example, here is x.properties in php.py:

```
# properties for mode php.xml
properties = {
    "commentEnd": "-->",
    "commentStart": "<!--",
    "indentCloseBrackets": "}",
    "indentOpenBrackets": "{",
    "lineUpClosingBracket": "true",
}
```

10.4.3 Attribute dictionaries and x.attributesDictDict

x.py contains a **attribute dictionary** for each ruleset in x.xml. Keys are attribute names, values strings representing the values of the attributes. This dictionary is empty if a ruleset contains no attributes. The valid keys are:

- `default': the default token type. `null' is the default.
- `digit_re': a regular expression. Words matching this regular expression are colored with the digit token type.
- `ignore_case': `true' or `false'. Default is `true'.
- `highlight_digits': `true' or `false'. Default is `true'.
- `no_word_sep': A list of characters treated as `alphabetic' characters when matching keywords.

For example, here is one attribute dictionary in php.py:

```
# Attributes dict for php_javascript ruleset.
php_javascript_attributes_dict = {
    "default": "MARKUP",
    "digit_re": "",
    "highlight_digits": "true",
    "ignore_case": "true",
    "no_word_sep": "",
}
```

x.py also contains **x.attributesDictDict**. Keys are ruleset names, values are attribute dictionaries. Here is attributesDictDict for php.py:

```
# Dictionary of attributes dictionaries for php mode.
attributesDictDict = {
    "php_javascript": php_javascript_attributes_dict,
    "php_javascript_php": php_javascript_php_attributes_dict,
    "php_main": php_main_attributes_dict,
    "php_php": php_php_attributes_dict,
    "php_php_literal": php_php_literal_attributes_dict,
    "php_phpdoc": php_phpdoc_attributes_dict,
    "php_tags": php_tags_attributes_dict,
    "php_tags_literal": php_tags_literal_attributes_dict,
}
```

Note: The jEdit2Py script creates `friendly' names for attribute dictionaries *solely* as an aid for people reading the code. Leo's colorer uses only the name x.attributeDictDict; Leo's colorer never uses the actual names of attribute dictionaries.

10.4.4 Keyword dictionaries and x.keywordsDictDict

x.py contains a **keyword dictionary** for each ruleset in x.xml. x.py contains an empty keywords dictionary if a ruleset contains no <KEYWORDS> element.

Keys are strings representing keywords of the language describe by the mode. Values are strings representing syntactic categories, i.e. a TYPE attribute valid in x.xml, namely: COMMENT1, COMMENT2, COMMENT3, COMMENT4, FUNCTION, KEYWORD1, KEYWORD2, KEYWORD3, KEYWORD4, LABEL, LITERAL1, LITERAL2, LITERAL3, LITERAL4, MARKUP, NULL and OPERATOR.

For example, here (parts of) some keyword dictionaries in php.py:

```
# Keywords dict for mode php::PHP
php_PHP_keywords_dict = {
    "COM_invoke": "keyword2",
    "COM_load": "keyword2",
    "__CLASS__": "keyword3",
    ...
    "abs": "keyword2",
    "abstract": "keyword1",
    "accept_connect": "keyword2",
    ...
}

# Keywords dict for mode php::JAVASCRIPT_PHP
php_JAVASCRIPT_PHP_keywords_dict = {}

# Keywords dict for mode php::PHPDOC
php_PHPDOC_keywords_dict = {
    "@abstract": "label",
    "@access": "label",
    "@author": "label",
    ...
    "@var": "label",
    "@version": "label",
}
```

x.py also contains **x.keywordsDictDict**. Keys are ruleset names, values are keywords dictionaries. Here is keywordsDictDict for php.py:

```
# Dictionary of keywords dictionaries for php mode.
keywordsDictDict = {
    "php_javascript": php_javascript_keywords_dict,
    "php_javascript_php": php_javascript_php_keywords_dict,
    "php_main": php_main_keywords_dict,
    "php_php": php_php_keywords_dict,
    "php_php_literal": php_php_literal_keywords_dict,
    "php_phpdoc": php_phpdoc_keywords_dict,
    "php_tags": php_tags_keywords_dict,
    "php_tags_literal": php_tags_literal_keywords_dict,
}
```

The colorizer can get the keywords dictionary for a ruleset as follows:

```
keywordsDict = x.keywordsDictDict(rulesetName)
```

Note: The jEdit2Py script creates `friendly' names for keyword dictionaries *solely* as an aid for people reading the code. Leo's colorer uses only the name x.keywordsDictDict; Leo's colorer never uses the actual names of keywords dictionaries such as php_PHPDOC_keywords_dict.

10.4.5 Rules, rules dictionaries and x.rulesDictDict

x.py contains one **rule function** for every rule in every ruleset (<RULES> element) in x.xml. These rules have names rule1 through ruleN, where N is the total number of rules in all rulesets in x.xml.

Each rules *function* merely calls a rules *method* in Leo's colorizer. Which method gets called depends on the corresponding element in *x.xml*. For example, the first rule in php.xml is:

```
<SPAN TYPE="MARKUP" DELEGATE="PHP">
        <BEGIN>&lt;?php</BEGIN>
        <END>?&gt;</END>
    </SPAN>
```

and the corresponding rule function is:

```
def php_rule0(colorer, s, i):
    return colorer.match_span(s, i, kind="markup", begin="<?php", end="?>",
        at_line_start=False, at_whitespace_end=False, at_word_start=False,
        delegate="PHP",exclude_match=False,
        no_escape=False, no_line_break=False, no_word_break=False)
```

php_rule0 calls colorer.match_span because the corresponding xml rule is a <SPAN> element.

For each ruleset, x.py also contains a **rules dictionary**, a Python dictionary whose keys are characters and whose values are all lists of rules functions that that can match the key. For example:

```
# Rules dict for phpdoc ruleset.
rulesDict8 = {
    "*": [rule64,],
    "0": [rule70,],
    "1": [rule70,],
    "2": [rule70,],
    "3": [rule70,],
    "4": [rule70,],
    "5": [rule70,],
    "6": [rule70,],
    "7": [rule70,],
    "8": [rule70,],
    "9": [rule70,],
    "<": [rule65,rule66,rule67,rule68,rule69,],
    "@": [rule70,],
    "A": [rule70,],
    "B": [rule70,],
    ...
    "X": [rule70,],
    "Y": [rule70,],
    "Z": [rule70,],
    "_": [rule70,],
    "a": [rule70,],
    "b": [rule70,],
   ...
    "x": [rule70,],
    "y": [rule70,],
    "z": [rule70,],
    "{": [rule63,],
}
```

Note: The order of rules in each rules list is important; it should be the same as rules element in x.xml.

Finally, x.py contains **x.rulesDictDict**. Keys are ruleset names, values are rules dictionaries. The colorer can get the rules list for character ch as follows:

```
self.rulesDict = x.rulesDictDict.get(rulesetName) # When a mode is inited.
...
rules = self.rulesDict.get(ch,[]) # In the main loop.
```

For example, here is the rules dictionary for php.py:

```
# x.rulesDictDict for php mode.
rulesDictDict = {
    "php_javascript": rulesDict6,
    "php_javascript_php": rulesDict7,
    "php_main": rulesDict1,
    "php_php": rulesDict4,
    "php_php_literal": rulesDict5,
    "php_phpdoc": rulesDict8,
    "php_tags": rulesDict2,
    "php_tags_literal": rulesDict3,
}
```

Note: The jEdit2Py script creates `friendly' names for rules lists *solely* as an aid for people reading the code. Leo's colorer uses only the name x.rulesDictDict; Leo's colorer never uses the actual names of rules lists such as rulesDict8, and Leo's colorer never uses the actual names of rules functions such as rule64.

10.4.6 x.importDict and imported versus delegated rulesets

x.importDict is a Python dictionary. Keys are ruleset names; values are a list of ruleset names. For example:

```
# Import dict for php mode.
importDict = {
    "php_javascript_php": ["javascript::main"],
}
```

For any ruleset R whose ruleset name is N, x.importDict.get(N) is the list of rulesets names whose rulesets appear in a DELEGATE attribute of an <IMPORT> rule element in R's ruleset. Such **imported** ruleset are copied to the end of the R's rules list. Leo's colorizer does this copying only once, when loading ruleset R for the first time.

Note 1: Loading imported rulesets must be done at `run time'. It should definitely not be done by jEdit2Py at `compile time'; that would require running jEdit2Py on *all* .xml files whenever any such file changed.

Note 2: Multiple <IMPORT> rule elements in a single ruleset are allowed: delegated rules are copied to the end of N's rules list in the order they appear in the ruleset.

Note 3: The DELEGATE attribute of <IMPORT> elements is, in fact, completely separate from the DELEGATE attributes of other rules as discussed in Arguments to rule methods. Indeed, the DELEGATE attribute of <IMPORT> elements creates entries in x.importDict, which in turn causes the colorizer to append the rules of the imported ruleset to the end of the present rules list. In contrast, the DELEGATE attributes of other rules sets the delegate argument to rules methods, which in tern causes the colorizer to recursively color the matched text with the **delegated** ruleset. In short:

- The rules of **imported** rulesets are appended to the end of another rules list; the rules of **delegated** rulesets never are.
- **Imported** ruleset names appear as the values of items in x.importDict; **delegated** ruleset names appear as delegate arguments to rule methods.

10.5 Rule methods

This section describes each rules method in Leo's new colorizer. Rules methods are called by rules functions in colorizer control file; they correspond directly to rules elements in jEdit's language description files. In fact, this documentation is a `refactoring' of jEdit's documentation.

All rule methods attempt to match a pattern at a particular spot in a string. These methods all return True if the match succeeds.

10.5.1 Arguments to rule methods

All rule methods take three required arguments and zero or more optional keyword arguments.

Here is a list of the required arguments and their meaning:

- **self**: An instance of Leo's colorizer.
- **s**: The string in which matches may be found.
- **i**: The location within the string at which the rule method looks for a match.

Here is a list of all optional keyword arguments and their meaning:

- **at_line_start**: If True, a match will succeed only if i is at the start of a line.
- **at_whitespace_end**: If True, the match will succeed only if i is at the first non-whitespace text in a line.
- **at_word_start**: If True, the match will succeed only if i is at the beginning of a word.
- **delegate**: If non-empty, the value of this argument is a ruleset name. If the match succeeds, the matched text will be colored recursively with the indicate ruleset.
- **exclude_match**: If True, the actual text that matched will not be colored. The meaning of this argument varies slightly depending on whether one or two sequences are matched. See the individual rule methods for details.
- **kind**: A string representing a class of tokens, i.e., one of: `comment1', `comment2', `comment3', `comment4', `function', `keyword1', `keyword2', `keyword3', `keyword4', `label', `literal1', `literal2', `literal3', `literal4', `markup', `null' and `operator'.
- **no_escape**: If True, the ruleset's escape character will have no effect before the end argument to match_span. Otherwise, the presence of the escape character will cause that occurrence of the end string to be ignored.
- **no_line_break**: If True, the match will not succeed across line breaks.
- **no_word_break**: If True, the match will not cross word breaks.

New in Leo 4.4.1 final: the regular expression rule matchers no longer get a hash_char argument because such matchers are called only if the present search pattern starts with hash_char.

10.5.2 match_eol_span

```
def match_eol_span (self,s,i,kind,begin,
    at_line_start = False,
    at_whitespace_end = False,
    at_word_start = False,
    delegate = '',
    exclude_match = False):
```

match_eol_span succeeds if s[i:].startswith(begin) and the at_line_start, at_whitespace_end and at_word_start conditions are all satisfied.

If successful, match_eol_span highlights from i to the end of the line with the color specified by kind. If the exclude_match argument is True, only the text before the matched text will be colored. The delegate argument, if present, specifies the ruleset to color the colored text.

10.5.3 match_eol_span_regexp

```
def match_eol_span_regexp (self,s,i,kind,regex,
    at_line_start = False,
    at_whitespace_end = False,
    at_word_start = False,
    delegate = '',
    exclude_match = False):
```

match_eol_span_exp succeeds if:

1. The regular expression regex matches at s[i:], and
2. The at_line_start, at_whitespace_end and at_word_start conditions are all satisfied.

If successful, match_eol_span_regexp highlights from i to the end of the line. If the exclude_match argument is True, only the text before the matched text will be colored. The delegate argument, if present, specifies the ruleset to color the colored text.

10.5.4 match_keywords

```
def match_keywords (self,s,i):
```

match_keywords succeeds if s[i:] starts with an identifier contained in the mode's keywords dictionary d.

If successful, match_keywords colors the keyword. match_keywords does not take a kind keyword argument. Instead, the keyword is colored as specified by d.get(theKeyword).

10.5.5 match_mark_following

```
def match_mark_following (self,s,i,kind,pattern,
    at_line_start = False,
    at_whitespace_end = False,
    at_word_start = False,
    exclude_match = False):
```

match_mark_following succeeds if s[i:].startswith(pattern), and the at_line_start, at_whitespace_end and at_word_start conditions are all satisfied.

If successful, match_mark_following colors from i to the start of the next token with the color specified by kind. If the exclude_match argument is True, only the text after the matched text will be colored.

10.5.6 match_mark_previous

```
def match_mark_previous (self,s,i,kind,pattern,
   at_line_start = False,
   at_whitespace_end = False,
   at_word_start = False,
   exclude_match = False):
```

match_mark_previous succeeds if s[i:].startswith(pattern),and the at_line_start, at_whitespace_end and at_word_start conditions are all satisfied.

If successful, match_mark_previous colors from the end of the previous token to i with the color specified by kind. If the exclude_match argument is True, only the text before the matched text will be colored.

10.5.7 match_seq

```
def match_seq (self,s,i,kind,seq,
   at_line_start = False,
   at_whitespace_end = False,
   at_word_start = False,
   delegate = ''):
```

match_seq succeeds if s[i:].startswith(seq) and the at_line_start, at_whitespace_end and at_word_start conditions are all satisfied.

If successful, match_seq highlights from i to the end of the sequence with the color specified by kind. The delegate argument, if present, specifies the ruleset to color the colored text.

10.5.8 match_seq_regexp

```
def match_seq_regexp (self,s,i,kind,regex,
   at_line_start = False,
   at_whitespace_end = False,
   at_word_start = False,
   delegate = ''):
```

match_seq succeeds if:

1. The regular expression regex matches at s[i:], and
2. The at_line_start, at_whitespace_end and at_word_start conditions are all satisfied.

If successful, match_seq_regexp highlights from i to the end of the sequence with the color specified by kind. The delegate argument, if present, specifies the ruleset to color the colored text.

10.5.9 match_span

```
def match_span (self,s,i,kind,begin,end,
    at_line_start = False,
    at_whitespace_end = False,
    at_word_start = False,
    exclude_match = False,
    delegate = ''
    no_escape = False,
    no_line_break = False,
    no_word_break = False):
```

match_span succeeds if there is an index j > i such that s[:i].startswith(begin) and s[i:j].endswith(end) and the at_line_start, at_whitespace_end, at_word_start, no_escape, no_line_break and no_word_break conditions are all satisfied.

If successful, match_span highlights from s[i:j with the color specified by kind; but if the exclude_match argument is True, the begin and end text are not colored. The delegate argument, if present, specifies the ruleset to color the colored text.

10.5.10 match_span_regexp

```
def match_span (self,s,i,kind,regex,end,
    at_line_start = False,
    at_whitespace_end = False,
    at_word_start = False,
    exclude_match = False,
    delegate = ''
    no_escape = False,
    no_line_break = False,
    no_word_break = False):
```

match_span_regex succeeds if:

1. The regular expression regex matches at s[i:],
2. There is an index j > i such that s[i:j].endswith(end),
3. The at_line_start, at_whitespace_end, at_word_start, no_escape, no_line_break and no_word_break conditions are all satisfied.

If successful, match_span colors s[i:j], with the color specified by kind; but if the exclude_match argument is True, the begin and end text are not colored. The delegate argument, if present, specifies the ruleset to color the colored text.

10.5.11 match_terminate

```
def match_terminate (self,s,i,kind,at_char):
```

match_terminate succeeds if s[i:] contains at least at_char more characters.

If successful, match_terminate colors at_char characters with the color specified by kind.

第 **11** 章

Creating Documents with Leo

This chapter discusses Leo's rst3 command. The rst3 command creates HTML, PDF, LaTeX and other kinds of documents from Leo outlines containing reStructuredText (rST) or Sphinx markup.

Note: docutils is a document processing system using rST markup. Sphinx extends docutils: Sphinx markup is a superset of rST markup. All of Leo's documentation is written in Sphinx and processed with the rst3 command to produce HTML files.

Leo and rst3 make writing rST/Sphinx documents a *lot* easier:

1. Leo outlines organize writing in all the usual ways. You always see the structure of your writing clearly no matter how large it is. You can reorganize chapters, sections and paragraphs effortlessly. View nodes can show you many different views of your writing simultaneously. These features, all by themselves, would make Leo an excellent choice for editing rST documents.
2. But the rst3 command does more: it automatically creates rST sections from headlines. Without Leo, changing the level of a section is clumsy and error prone: you must carefully change the section's underlining character. Leo's rst3 command eliminates all this bother.
3. The rst3 command converts an @rst tree to rST and then sends this text to docutils or Sphinx for further processing.

In addition to these basic features, the rst3 command provides *many* other capabilities for power users. We'll discuss these features later.

Contents

- Creating Documents with Leo
  - Tutorial
    - Step 1: Create the @rst node
      - Set global options
      - Set your document's title
    - Step 2: Write your document
    - Step 3: Run the rst3 command
    - Go forth and experiment
  - Options
    - General options
    - Headline commands
    - Option doc parts
    - Defaults for options
    - Http plugin options
    - Section expansion options
  - Other topics
    - Markup doc parts
    - Required cascading style sheets
  - Further study
  - Acknowledgements
  - Theory of operation

11.1 Tutorial

This tutorial tells how to get started with Leo's rst3 command. The tutorial covers only rst3's basic features. You can do a *lot* with these features--Leo's documentation uses only the features described here!

Step-by-step, here is how to use the rst3 command:

1. Create an @rst node. This node and its descendants will contain your document. The @rst node itself is a good place to specify general information about your documentation, including its title, one or more external files created by the rst3 command, and global settings for the rst3 command.
2. Write your documentation in the descendants of the @rst node. Within the @rst tree, **headlines represent section headings**. Body text contain your writing, including rST or Sphinx markup.
3. To create your documents, run the rst3 command on an outline containing one or more @rst nodes.

That's all there is to it! The organization of the @rst tree *is* the organization of your document. To reorganize your document, you just reorganize the nodes in the @rst tree! When you are done writing, create your output using the rst3 command.

The next sections will discuss these three steps in more detail. As you will see, after you set up the @rst node, you can focus exclusively on writing and organizing. Leo's rst3 command will take care of the rest.

11.1.1 Step 1: Create the @rst node

The headline of the @rst node has the form:

```
@rst <filename>
```

Depending on options to be discussed later, the rst3 command will write one or two files: the **output file** (<filename>), and the **intermediate file** (<filename>.txt).

For example, the rst3 command applied to @rst abc.html will write abc.html or abc.html.txt or both.

Important: The intermediate file *always* contains rST/Sphinx markup *regardless* of the type of the final output files. When in doubt about the rst3 command, you can examine the intermediate file to see what rst3 has done.

Let's turn our attention to the the body of the @rst node...

Set global options

The @rst node is a good place for options that apply to the entire @rst tree. Typically, you will just set these options once and then completely forget about them.

You set rst3 options in body parts like this:

```
@ @rst-options
rst3 options, one per line
@c
```

This is a Leo doc part: the `@' must appear in the leftmost column. As usual, the doc part ends with the @c directive, or the end of the body text.

Here are the recommended options when using docutils:

```
@ @rst-options
call_docutils=True
code_mode=False
stylesheet_path=..\doc
write_intermediate_file=True
@c
```

And here are the recommended options when using Sphinx:

```
@ @rst-options
call_docutils=False
code_mode=False
stylesheet_path=..\doc
write_intermediate_file=True
@c
```

Note 1: It is good style to specify all these options explicitly, even if they are the same as the standard default values. This ensures that the rst3 command will produce the same results no matter where the @rst node is located.

Note 2: You may have to change the stylesheet_path option so that the generated output file can find the proper stylesheets.

Set your document's title

Next, set your document's title by putting something like this in the body text of the @rst node:

```
#############
War and Peace
#############
```

Important: The rst3 command reserves the `#' character for the document titles--don't use any other underlining character.

Sometimes I put the first words of a document in the @rst node:

```
Well, Prince, so Genoa and Lucca are now just family estates of the
Buonapartes. But I warn you, if you don't tell me that this means war, if
you still try to defend the infamies and horrors perpetrated by that
Antichrist--I really believe he is Antichrist--I will have nothing more to
do with you and you are no longer my friend, no longer my 'faithful slave,'
as you call yourself! But how do you do? I see I have frightened you--sit
down and tell me all the news.
```

11.1.2 Step 2: Write your document

Now comes the ``interesting'' part--actually writing your novel, short story, documentation or whatever.

As always with Leo, you organize your work with outlines. By default, (that is, with the recommended options discussed in Step 1) the rst3 command will produce the following output:

1. Each node becomes an rST/Sphinx section.

   The level of each section corresponds to the level of the node in the headline. Children of the @rst node create level 1 sections. Grandchildren of the @rst node create level 2 sections, and so on.

2. The headline of each node becomes the section heading.

3. The body text of each node becomes the contents of the node's section.

   Note: The body text of any node in an @rst tree contains plain text, with optional rST or Sphinx markup. Sphinx markup is a superset of rST markup. For more details on markup, see the Sphinx or reStructuredText documentation.

That's all there is to it!

Well almost. There is one other feature you should know about. Headlines that start with `@rst-` control the rst3 command. The three most useful are:

@rst-no-head <ignored-text>

> Causes the rst3 command to copy just the body text of the node. In other words, the node's body text become part of the previous section. Leo's docs use such nodes for rST links and other ``invisible'' markup.

@rst-ignore <ignored-text>

> The rst3 command ignores any @rst-ignore node. Neither the headline nor the body text becomes part of the output. You can use such nodes for notes that you do not want to become part of the actual document.

@rst-ignore-tree <ignored-text>

The rst3 command ignores the @rst-ignore-tree node and all its descendants.

11.1.3 Step 3: Run the rst3 command

This step is easy. Select an outline containing one or more @rst trees. Now do <Alt-X>rst3<Return>. You can use <Ctrl-P> (repeat-complex-command) instead if the last minibuffer command was the rst3 command.

The rst3 command writes its output to either the output file or the intermediate file, or both:

- With the recommended settings for docutils, the rst3 command will run docutils automatically, producing the output file as the result.
- With the recommended settings for Sphinx, the rst3 command will generate the intermediate file. You must then run Sphinx's make utility to turn the intermediate file into the final output file.

11.1.4 Go forth and experiment

You now know everything needed to get started with the rst3 command. Some possible next steps are:

1. Look at Leo's own documentation in LeoDocs.leo. It's in the node ``@rst htmlrstplugin3.html". Discover how the nodes in this tree correspond to the documentation you see before you.
2. Create your own @rst nodes. Run the rst3 command on them and see what happens. If you get stuck, you please ask for help at Leo's Google Group.

11.2 Options

This section discusses options--what they are, how to set them and how to set their defaults.

11.2.1 General options

Here is a complete list of options for the rst3 and code-to-rst commands:

call_docutils (default: True):

Call docutils to process the intermediate file.

default_path (default: `')

The path to be prepended to filenames given in root nodes.

default_encoding (default: utf-8)

The default encoding to be used for non-ascii (unicode characters).

encoding (default: the default_encoding setting)

The encoding to be used for non-ascii (unicode) characters. **Important**: this option has effect only in @rst-options doc parts in root @rst nodes.

generate_rst (default: True)

A master switch. True: generate rST markup for rST sections and rST code-blocks. False: generate plain text and ignore @ @rst-markup doc parts.

generate_rst_header_comment (default: True)

True: Leo writes a comment line of the form:

```
.. rst3: filename: <filename>
```

at the start of intermediate files. This option has effect only if the generate_rst and write_intermediate_file options are both True.

publish-argv-for-missing-stylesheets (Default: `')

The arguments to be passed to docutils.core.Publisher().publish() when no stylesheet is in effect. This is a string that represents a comma-separated list of strings: For example, the option:

```
publish-argv-for-missing-stylesheets=--language=de,--documentclass=report,--use-latex-toc
```

results in the call:

```
publish(['--language=de','--documentclass=report','--use-latex-toc'])
```

show_headlines (default: True)

True: automatically generate rST sections from headlines. False: ignore headlines.

Note: The level of the node in the outline determines the level of the section underlining in the rST markup. Higher-level headlines in the outline correspond to higher-level section headings; lower-level headlines in the outline correspond to lower-level section headings.

show_organizer_nodes (default: True)

True: generate rST sections for nodes that do not contain body text.

Note: This option has no effect unless the rST section would otherwise be written.

show_sections (default: True)

True: generate rST sections corresponding to headlines. False: don't generate sections. Instead, generate lines of the form:

```
**headline**
```

strip_at_file_prefixes (default: True)

True: remove @auto, @file, @nosent and @thin from the start of headlines.

stylesheet_name (default: `default.css')

The name of the stylesheet passed to docutils.

stylesheet_path (default: `')

The directory containing the stylesheet passed to docutils.

Note: If the stylesheet_embed option is True, specify a path relative to the location of the Leo file. If the stylesheet_embed option is False, specify a path relative to the location of the HTML file.

stylesheet_embed (default: True)

True: The content of the stylesheet file will be embedded in the HTML file. False: The HTML file will link to an external stylesheet file.

underline_characters (default: #=+*^~"'`-:>_)

The underlining characters to be used to specify rST sections. The first character is reserved so you can specify the top-level section explicitly.

verbose (default: True)

True: write informational messages.

write_intermediate_file (default: False)

Important: the rst3 command *always* creates an intermediate file. This option controls whether that file is an internal Python object or an actual file on the external file system.

True: writes the intermediate file to the external file system. The name of the intermediate file has the name of the output file with .txt appended. This option has effect only if the generate_rst option is True.

False: writes the intermediate file to an internal Python object.

11.2.2 Headline commands

Any headline that starts with @rst- controls the rst3 command.

@rst-ignore <ignored-text>

Ignore the node and its descendants.

@rst-ignore-node <ignored-text>

Ignore the node, but *not* its descendants.

@rst-ignore-tree <ignored-text>

Same as @rst-ignore. Ignore the node and its descendants.

@rst-no-head <ignored-text>

Ignore the headline but not the body text of this node. This has no effect on descendant nodes.

@rst-no-headlines <ignored-text>

Ignore all headlines. (Set show_headlines=False)

@rst-option <option> = <value>

Set a single option to the given value. The default value is True.

@rst-options <ignored-text>

Set options from body text. The body text should contain nothing but lines of the form:

```
<option>=<value>
```

@rst-preformat <ignored-text>

Format the body text of the node as computer source code. In effect, this option adds a line containing `::' at the start of the body text. The option then indents all following lines.

This option has no effect on descendant nodes.

11.2.3 Option doc parts

Option doc parts set rst3 options. Option doc parts start with @ @rst-options followed by lines of the form name=value. (Comment lines starting with `..' are allowed.) For example:

```
@ @rst-options
.. This comment line is ignored.
show_headlines=False
show_leo_directives=False
verbose=True
@c
```

This is a real Leo doc part. Like all other doc parts an option doc part starts with the @ directive and continues until the end of body text or until the next @c directive.

11.2.4 Defaults for options

Settings in leoSettings.leo or myLeoSettings.leo specify the defaults to be used for all rst3 options. The form of these settings is:

```
@bool rst3_<option name> = True/False
@string rst3_<option name> = aString
```

That is, to create a default value for an rst3 setting, you must prefix the option name with `rst3_'. For example:

```
@bool rst3_write_intermediate_file = True
```

11.2.5 Http plugin options

The following options are for the use of Bernhard Mulder's http plugin. The http plugin creates an http server running on a local port, typically 8080. When the http plugin is running you will see a purple message in the log window that looks something like this:

```
http serving enabled on port 8080, version 0.91
```

To use the http plugin, start a web browser and enter this url:

```
http://localhost:8080/
```

You will see a a top level page containing one link for every open .leo file. Clicking on a link will cause the http server to pass a new page to the browser. You can use the browser's refresh button to update the top-level view in the browser after you have opened or closed files.

Important: See the docstring for the http plugin for information on configuring the plugin. Some of the following rst3 settings must match values of settings for the http plugin.

Here are the rst3 options that support the http plugin:

http_server_support (default: False)

A master switch: none of the following options have any effect unless this option is True. If True, the rst3 command does the following:

1. Writes **node markers** in the rst output for use by the http plugin. Node markers are rst named hyperlink targets. By default they look like:

```
.. _http-node-marker-N
```

   where N is a unique node number.

2. Adds additional information to all nodes of the tree being formatted using Leo's unknownAttributes mechanism.

http_attributename (default: `rst_http_attribute')

The name of the attribute name written to the unknownAttributes attribute of each outline node in the rst root tree. The default is `rst_http_attribute'; it should match the following setting of the http plugin:

```
@string rst_http_attributename = 'rst_http_attribute'
```

clear_http_attributes (default: False)

If True the rst3 command initially clears the fields specified by *http_attributename*.

node_begin_marker (default: `http-node-marker-`)

The string used for node markers.

11.2.6 Section expansion options

New in Leo 4.9: The following three options allow you to expand noweb section references, much like Leo itself does.

expand_noweb_references

True: Replace references by definitions. Definitions must be descendants of the referencing node.

ignore_noweb_definitions

True: ignore section definition nodes.

expand_noweb_recursively

True: recursively expand definitions by expanding any references found in definitions.

Notes:

- This is an experimental feature: all aspects might changed. The defaults for all these options ensure that the rst3 command works as it has always.
- The rst3 command ensures that unbounded expansions can not happen. While expanding any section, the rst3 will not expand again any sections that have already occurred in the expansion.

11.3 Other topics

11.3.1 Markup doc parts

Markup doc parts have the following form:

```
@ @rst-markup
any rST markup
@c
```

Markup doc parts inserts the markup directly into the output. Markup doc parts are most useful when formatting an outline as code using the code-to-rst command.

11.3.2 Required cascading style sheets

HTML files generated by the rst3 command assume that three .css (cascading style sheet) files exist in the same directory. For the HTML output to look good the following .css files should exist:

- default.css is the default style sheet that docutils expects to exist.
- leo_rst.css contains some style improvements based on Gunnar Schwant's DocFactory.
- silver_city.css is the style sheet that controls the syntax highlighting generated by SilverCity.

The latter two style sheets are imported at the end of the default.css.

Important: You can use cascading style sheets to do things that otherwise wouldn't be possible with ``plain'' rST. For instance, the background color of this page was specified in a body style.

11.4 Further study

The file ListManagerDocs.html is an impressive example of the kind of output that can be generated relatively easily using the rst3 command.

The source for ListManagerDocs.html is wxListManager.leo. **Important**: wxListManager.leo was written for the old rst2 plugin; it could be greatly simplified if adapted for the rst3 command.

This documentation was created using the rst3 command. The source code for this documentation is in LeoDocs.leo. The source code for the rst3 command is in leoRst.py in leoPy.leo.

11.5 Acknowledgements

Josef Dalcolmo wrote the initial rst plugin. Timo Honkasalo, Bernhard Mulder, Paul Paterson, Kent Tenney and Steve Zatz made contributions to the rst and rst2 plugins.

11.6 Theory of operation

The code for the rst3 command is more complex than usual. Fortunately, the overall organization is straightforward.

defaultOptionsDict This dictionary represents each rst3 option. To add another option, just add another entry to this dictionary. Keys are the option name, including the *rst3_* prefix. Values are the default value of the option. The hard-coded values in the dictionary may be changed as the result of @settings entries.

processTree processTree is the top-level code that handles one rst root node. It calls preprocessTree to create the **vnodeOptionDict** ivar. processTree then calls either writeNormalTree or writeSpecialTree depending on whether text will be sent to docutils for further processing. These two methods handle mundane details of opening and closing files. Both writeNormalTree and writeSpecialTree call **writeTree** to do the actual work.

vnodeOptionDict The entries in this dictionary represent the options that are set in one particular node. The keys of vnodeOptionDict are vnodes, the values are anonymous dictionaries. These anonymous inner dictionaries contain the options that are explicitly set at each vnode (and thus each position). Preprocessing the tree this way ensures that each node (headline and body text) is parsed exactly once.

writeTree writeTree first calls **scanAllOptions**, which has the effect of initializing all options. writeTree then calls **writeNode** for each node that will be processed by the rst3 command. Options may cause the rst3 command to skip a node or an entire subtree.

writeNode writeNode first calls **scanAllOptions** to compute the options that are in effect for that *single* node. Once options have been computed, processing the node is straightforward. writeNode calls writeBody and writeHeadline to do the real work. These methods generate or skip text based on various options.

scanAllOptions scanAllOptions recreates the optionsDict ivar to represent *all* the options in effect for *for the particular node being processed by writeNode*. Client code gets these options by calling the getOption method.

scanAllOptions first inits all options from settings, then updates those options using the anonymous dictionaries contained in the vnodeOptionsDict. scanAllOptions works like g.scanAllDirectives, but the code is much simpler.

第 **12** 章

Scripting Leo with Python

This chapter describes how to write Python scripts that control Leo and access the data in Leo outlines. To write such scripts, you must understand the basics of Leo's internal data structures. As we shall see, these basics are quite simple.

Although this chapter discusses everything you will need to write most scripts, please keep in mind that your scripts have complete access to all of Leo's source code, that is, all the code in LeoPy.leo.

You can put Leo script in the body text of *any* outline node. You can run scripts in several ways:

1. Select a node containing the script and do Ctrl-B (execute-script).
2. @button and @command nodes allow you to run scripts while selecting some *other* node. This brings scripts to data, an extremely powerful capability. @button nodes create a button in Leo's icon area. You can execute the script **on the presently selected node** by pressing the button. @command nodes work similarly, but instead of creating a button, they create a Leo command. (Actually, @button nodes also create Leo commands).

Contents

- Scripting Leo with Python
  - Basics
    - c, g and p
    - Import objects
    - g.es writes to the log pane
    - p.h and p.b
    - c.redraw
    - p.copy
    - Generators
    - Testing whether a position is valid
    - g.pdb
    - @button scripts
    - autocompletion
    - Summary
  - Event handlers
    - Enabling idle time event handlers
  - Other topics
    - g.app.windowList: the list of all open frames
    - Ensuring that positions are valid
    - g.openWithFileName
    - g.getScript
    - c.frame.body.bodyCtrl
    - Invoking commands from scripts
    - Getting settings from @settings trees
    - Preferences ivars
    - Functions defined in leoGlobals.py
    - Making operations undoable
    - Redirecting output from scripts
    - Creating Qt Windows from Leo scripts
    - Writing to different log tabs
    - Invoking dialogs using the g.app.gui class
    - Inserting and deleting icons
    - Working with directives and paths
    - Running Leo in batch mode
    - Getting interactive input from scripts
    - The @g.command decorator
    - Modifying plugins with @script scripts
    - Creating minimal outlines
    - Retaining pointers to Qt windows

12.1 Basics

12.1.1 c, g and p

All Leo scripts run with the execute-script command (Ctrl-B) have access to the following three **predefined objects**:

- c is the commander of the outline containing the script.
- g is Leo's leo.core.leoGlobals module.
- p is the presently selected position, the same as c.p.

The predefined c, g and p variables give *full* and *easy* access to all the data in your outlines, as well as hundreds, if not thousands, of useful Python functions in Leo's core. These variables define an easy-to-use DOM (Document Object Model) through which you can get o set any data in *your* outlines. The hello world example is:

```
for p in c.all_positions():
    indent = '.' * p.level()
    print('%s%s' % (indent,p.h))
```

Here, c.all_positions() delivers a list of positions, and p.level() and p.h deliver the indentation level and head string of position p.

12.1.2 Import objects

Leo scripts typically use the following objects:

g The predefined object g is the leo.core.leoGlobals module. This module contains several dozen utility functions and classes.

g.app g.app is the **application object** representing the entire Leo application. The instance variables (ivars) of g.app represent Leo's global variables.

commander The predefined object c is the commander of the window containing the script. Commanders represent all aspects of a single Leo window. For any commander c, c.p is the presently selected position (see below), and c.rootPosition() is the root (first) position in the outline. Given c, Leo scripts can gain access to all data present while Leo is running, including all of Leo's classes, functions and data.

position The predefined object p is the position of the presently selected node. Positions represent locations in Leo outlines. For any position p, p.v is the vnode at that position.

vnode A vnode represents a single outline node. Because of clones, a vnode may appear in several places on the screen. Vnodes hold most of the data in Leo outlines. For any vnode v, v.h is the node's headline, and v.b is the node's body text. As a convenience, for any position p, p.h and p.b are synonyms for p.v.h and p.v.b.

Most scripts will need only the objects and classes described above.

12.1.3 g.es writes to the log pane

The g.es method prints its arguments to the Log tab of the log pane:

```
g.es("Hello world")
```

g.es converts non-string arguments using repr:

```
g.es(c)
```

g.es prints multiple arguments separated by commas:

```
g.es("Hello","world")
```

To create a tab named `Test' or make it visible if it already exists:

```
c.frame.log.selectTab('Test')
```

When first created, a tab contains a text widget. To write to this widget, add the tabName argument to g.es:

```
g.es('Test',color='blue',tabName='Test')
```

12.1.4 p.h and p.b

Here is how to access the data of a Leo window:

```
g.es(p) # p is already defined.
p = c.p # get the current position.
g.es(p)
g.es("head:",p.h)
g.es("body:",p.b)
```

Here is how to access data at position p. **Note**: these methods work whether or not p is the current position:

```
body = p.b # get the body text.
head = p.h # get the headline text.
p.b = body # set body text of p to body.
p.h = head # set headline text of p to head.
```

Note: Sometimes you want to use text that *looks* like a section reference, but isn't. In such cases, you can use g.angleBrackets. For example:

```
g.es(g.angleBrackets('abc'))
```

12.1.5 c.redraw

You can use c.redraw_now to redraw the entire screen immediately:

```
c.redraw_now()
```

However, it is usually better to *request* a redraw to be done later as follows:

```
c.redraw()
```

Leo actually redraws the screen in c.outerUpdate, provided that a redraw has been requested. Leo will call c.outerUpdate at the end of each script, event handler and Leo command.

12.1.6 p.copy

Scripts must wary of saving positions because positions become invalid whenever the user moves, inserts or deletes nodes. It is valid to store positions **only** when a script knows that the stored position will be used before the outline's structure changes.

To store a position, the script must use the p.copy() method:

```
p2 = p.copy()   # Correct: p2 will not change when p changes later.
```

The following **will not work**:

```
p2 = p  # Wrong.  p2 will change if p changes later.
```

For example, the following creates a dictionary of saved positions:

```
d = {}
for p in c.all_positions():
    d[p.v] = p.copy()
```

12.1.7 Generators

Leo scripts can easily access any node of an outline with iterator. Leo's iterators return positions or nodes, one after another. Iterators do not return lists, but you can make lists from iterators easily. For example, the c.all_positions() iterator returns every position in c's tree, one after another. You can use the iterator directly, like this:

```
for p in c.all_positions():
    print(p.h)
```

It is seldom a good idea to capture positions in a list: positions become invalid whenever the outline changes. Using an invalid position could cause a (soft) crash in Leo. If you do want to capture positions temporarily, the proper way is to call p.copy(). Like this:

```
aList = [p.copy() for p in c.all_positions()]
```

The following will **not** work:

```
aList = list(c.all_positions())
```

because it is equivalent to:

```
aList = [p for p in c.all_positions()]
```

The problem is that the c.all_positions iterator uses a *single* position, and merely alters that position each time it is called. Thus, aList will contain multiple copies of an empty position!

However, the following *does* work:

```
aList = list(c.all_nodes())
```

In this case, aList will contain a list of all the *vnodes* in the outline.

c.all_positions & c.all_unique_positions

The c.all_positions generator returns a list of all positions in the outline. This script makes a list of all the nodes in an outline:

```
nodes = list(c.all_positions())
print("This outline contains %d nodes" % len(nodes))
```

The c.all_unique_positions generator returns a list of all unique positions in the outline. For each vnode v in the outline, exactly one position p is returned such that p.v == v.

This script prints the *distinct* vnodes of an outline:

```
for p in c.all_unique_positions():
    sep = g.choose(p.hasChildren(),'+','-')
    print('%s%s %s' % (' '*p.level(),sep,p.h))
```

p.children

The p.children generator returns a list of all children of position p:

```
parent = p.parent()
print("children of %s" % parent.h)
for p in parent.children():
    print(p.h)
```

p.parents & p.self_and_parents

The p.parents generator returns a list of all parents of position p, excluding p:

```
current = p.copy()
print("exclusive of %s" % (current.h),color="purple")
for p in current.parents():
    print(p.h)
```

The p.self_and_parents generator returns a list of all parents of position p, including p:

```
current = p.copy()
print("inclusive parents of %s" % (current.h),color="purple")
for p in current.self_and_parents():
    print(p.h)
```

p.siblings & p.following_siblings

The p.siblings generator returns a list of all siblings of position p:

```
current = c.p
print("all siblings of %s" % (current.h),color="purple")
for p in current.self_and_siblings():
    print(p.h)
```

The p.following_siblings generator returns a list of all siblings that follow position p:

```
current = c.p
print("following siblings of %s" % (current.h),color="purple")
for p in current.following_siblings():
    print(p.h)
```

p.subtree & p.self_and_subtree

The p.subtree generator returns a list of all positions in p's subtree, excluding p:

```
parent = p.parent()
print("exclusive subtree of %s" % (parent.h),color="purple")
for p in parent.subtree():
    print(p.h)
```

The p.self_and_subtree generator returns a list of all positions in p's subtree, including p:

```
parent = p.parent()
print("inclusive subtree of %s" % (parent.h),color="purple")
for p in parent.self_and_subtree():
    print(p.h)
```

12.1.8 Testing whether a position is valid

The tests:

```
if p:       # Right
if not p:   # Right
```

are the **only** correct ways to test whether a position p is valid. In particular, the following **will not work**:

```
if p is None:       # Wrong
if p is not None:   # Wrong
```

12.1.9 g.pdb

g.pdb() invokes Python pdb debugger. You must be running Leo from a console to invoke g.pdb().

g.pdb() is merely a convenience. It is equivalent to:

```
import pdb
pdb.set_trace()
```

The debugger_pudb.py plugin causes g.pdb() to invoke the full-screen pudb debugger instead of pdb. pudb works on Linux and similar systems; it does not work on Windows.

12.1.10 @button scripts

Creating an @button script should be your first thought whenever you want to automate any task. The scripting plugin, mod_scripting.py, must be enabled to use @button scripts.

When Leo loads a .leo file, the mod_scripting plugin creates a **script button** in Leo's icon area for every @button node in the outline. The plugin also creates a corresponding minibuffer command for each @button node. Pressing the script button (or executing the command from the minibuffer) applies the script in the @button node to the presently selected outline node.

In effect, each script button defines an instant command! The .leo files in Leo's distribution contain many @button nodes (many disabled), that do repetitive chores. Here is one, @button promote-child-bodies, from LeoDocs.leo:

```
'''Copy the body text of all children to the parent's body text.'''

# Great for creating what's new nodes.
result = [p.b]
b = c.undoer.beforeChangeNodeContents(p)
for child in p.children():
    if child.b:
        result.append('\n- %s\n\n%s\n' % (child.h,child.b))
    else:
        result.append('\n- %s\n\n' % (child.h))
p.b = ''.join(result)
c.undoer.afterChangeNodeContents(p,'promote-child-bodies',b)
```

This creates a fully undoable promote-child-bodies command.

Notes:

- Script buttons execute the **present** body text of the @button node. You can modify a script button's script at any time without having to recreate the script button. This makes script buttons ideal for prototyping code.
- You can bind keys to the commands created by script buttons. For example:

```
    @button my-button @key=Alt-8
```

- You can delete any script button by right-clicking on it.
- For more details, see the docstring of the mod_scripting plugin. If the plugin is enabled, you can see this string by choosing mod_scripting from Leo's Plugins menu.

12.1.11 autocompletion

Alt-1 (toggle-autocompleter) enables and disables Leo's autocompletion feature. Autocompletion is extremely useful for writing Leo scripts because it knows about all of Python's standard library modules and all of Leo's source code. **Important**: @language python must be in effect for autocompletion to work.

For example, with autocompletion enabled typing:

```
c.atF
```

will put the only possible completion in the body pane:

```
c.atFileCommands
```

Continuing to type:

```
.wr
```

will show you all of the write commands in leoAtFile.py:

```
write:method
writeAll:method
writeAllHelper:method
writeAtAutoNodes:method
writeAtAutoNodesHelper:method
writeAtShadowNodes:method
writeAtShadowNodesHelper:method
writeDirtyAtAutoNodes:method
writeDirtyAtShadowNodes:method
writeError:method
writeException:method
writeFromString:method
writeMissing:method
writeOneAtAutoNode:method
writeOneAtEditNode:method
writeOneAtShadowNode:method
writeOpenFile:method
writeVersion5:<class 'bool
writing_to_shadow_directory:<class 'bool
```

When a single completion is shown, typing `?' will show the docstring for a method. For example:

```
c.atFileCommands.write?
```

shows:

```
Write a 4.x derived file.
root is the position of an @<file> node
```

Using autocompletion effectively can lots of time when writing Leo scripts.

12.1.12 Summary

The following sections summarizes the most useful methods that your scripts can use.

Iterators

Here is the list of Leo's iterators:

```
c.all_nodes             # all vnodes in c.
c.all_unique_nodes      # all unique vnodes in c.
c.all_positions         # all positions in c.
c.all_unique_positions  # all unique positions in c.
```

```
p.children           # all children of p.
p.following_siblings    # all siblings of p that follow p.
p.nodes             # all vnodes in p's subtree.
p.parents           # all parents of p.
p.self_and_parents      # p and all parents of p.
p.siblings          # all siblings of p, including p.
p.subtree           # all positions in p's subtree, excluding p.
p.self_and_subtree      # all positions in p's subtree, including p.
p.unique_nodes        # all unique vnodes in p's subtree.
p.unique_subtree       # all unique positions in p's subtree.
```

Note: An iterator that returns **unique positions** is an iterator that returns a list of positions such that p.v == v at most once for any vnode v. Similarly, a generator that returns **unique nodes** is a generator that returns a list that contains any vnode at most once.

Note: The names given above are the recommended names for Leo's iterators. Leo continues to support the names of iterators used before Leo 4.7. These names typically end with the _iter suffix.

Getters

Here are the most useful getters of the vnode and position classes.

Returning strings:

```
p.b # the body string of p.
p.h # the headline string of p. A property.
```

Returning ints:

```
p.childIndex()
p.numberOfChildren()
p.level()
```

Returning bools representing property bits:

```
p.hasChildren()
p.isAncestorOf(v2) # True if v2 is a child, grandchild, etc. of p.
p.isCloned()
p.isDirty()
p.isExpanded()
p.isMarked()
p.isVisible()
p.isVisited()
```

Setters

Here are the most useful setters of the Commands and position classes. The following setters of the position class regardless of whether p is the presently selected position:

```
p.b = s  # Sets the body text of p.
p.h = s  # Sets the headline text of p.
```

Moving nodes:

```
p.moveAfter(v2)           # move p after v2
p.moveToNthChildOf(v2,n)  # move p to the n'th child of v2
p.moveToRoot(oldRoot)     # make p the root position.
                          # oldRoot must be the old root position if it exists.
```

The ``visited" bit may be used by commands or scripts for any purpose. Many commands use this bits for tree traversal, so these bits do not persist:

```
c.clearAllVisited() # Clears all visited bits in c's tree.
p.clearVisited()
p.setVisited()
```

12.2 Event handlers

Plugins and other scripts can register event handlers (also known as hooks) with code such as:

```
leoPlugins.registerHandler("after-create-leo-frame",onCreate)
leoPlugins.registerHandler("idle", on_idle)
leoPlugins.registerHandler(("start2","open2","command2"), create_open_with_menu)
```

As shown above, a plugin may register one or more event handlers with a single call to leoPlugins.registerHandler. Once a hook is registered, Leo will call the registered function' at the named **hook time**. For example:

```
leoPlugins.registerHandler("idle", on_idle)
```

causes Leo to call on_idle at ``idle" time.

Event handlers must have the following signature:

```
def myHook (tag, keywords):
    whatever
```

- tag is the name of the hook (a string).
- keywords is a Python dictionary containing additional information. The following section describes the contents of the keywords dictionary in detail.

Important: hooks should get the proper commander this way:

```
c = keywords.get('c')
```

The following table tells about each event handler: its name, when it is called, and the additional arguments passed to the hook in the keywords dictionary. For some kind of hooks, Leo will skip its own normal processing if the hook returns anything *other* than None. The table indicates such hooks with `yes' in the `Stop?' column.

Important: Ever since Leo 4.2, the v, old_v and new_v keys in the keyword dictionary contain *positions*, not vnodes. These keys are deprecated. The new_c key is also deprecated. Plugins should use the c key instead.

Event name (tag argument)	Stop?	When called	Keys in keywords dict
`after-auto'		after each @auto file loaded	c,p (note 13)
`after-create-leo-frame'		after creating any frame	c
`after-redraw-outline'		end of tree.redraw	c (note 6)
`before-create-leo-frame'		before frame.finishCreate	c
`bodyclick1'	yes	before normal click in body	c,p,v,event
`bodyclick2'		after normal click in body	c,p,v,event
`bodydclick1'	yes	before double click in body	c,p,v,event
`bodydclick2'		after double click in body	c,p,v,event
`bodykey1'	yes	before body keystrokes	c,p,v,ch,oldSel,undoType
`bodykey2'		after body keystrokes	c,p,v,ch,oldSel,undoType
`bodyrclick1'	yes	before right click in body	c,p,v,event
`bodyrclick2'		after right click in body	c,p,v,event
`boxclick1'	yes	before click in +- box	c,p,v,event
`boxclick2'		after click in +- box	c,p,v,event
`clear-all-marks'		after clear-all-marks command	c,p,v
`clear-mark'		when mark is set	c,p,v
`close-frame'		in app.closeLeoWindow	c
`color-optional-markup'	yes *	(note 7)	colorer,p,v,s,i,j,colortag (note 7)
`command1'	yes	before each command	c,p,v,label (note 2)
`command2'		after each command	c,p,v,label (note 2)
`create-optional-menus'		(note 8)	c (note 8)
`create-popup-menu-items'		in tree.OnPopup	c,p,v,event (new)
`draw-outline-box'	yes	when drawing +- box	tree,p,v,x,y
`draw-outline-icon'	yes	when drawing icon	tree,p,v,x,y
`draw-outline-node'	yes	when drawing node	tree,p,v,x,y
`draw-outline-text-box'	yes	when drawing headline	tree,p,v,x,y
`drag1'	yes	before start of drag	c,p,v,event
`drag2'		after start of drag	c,p,v,event
`dragging1'	yes	before continuing to drag	c,p,v,event
`dragging2'		after continuing to drag	c,p,v,event
`enable-popup-menu-items'		in tree.OnPopup	c,p,v,event
`end1'		start of app.quit()	None
`enddrag1'	yes	before end of drag	c,p,v,event
`enddrag2'		after end of drag	c,p,v,event
`headclick1'	yes	before normal click in headline	c,p,v,event
`headclick2'		after normal click in headline	c,p,v,event
`headrclick1'	yes	before right click in headline	c,p,v,event
`headrclick2'		after right click in headline	c,p,v,event
`headkey1'	yes	before headline keystrokes	c,p,v,ch (note 12)
`headkey2'		after headline keystrokes	c,p,v,ch (note 12)
`hoist-changed'		whenever the hoist stack changes	c

Continued on next page

Table 12.1 -- continued from previous page

Event name (tag argument)	Stop?	When called	Keys in keywords dict
`hyperclick1'	yes	before control click in hyperlink	c,p,v,event
`hyperclick2'		after control click in hyperlink	c,p,v,event
`hyperenter1'	yes	before entering hyperlink	c,p,v,event
`hyperenter2'		after entering hyperlink	c,p,v,event
`hyperleave1'	yes	before leaving hyperlink	c,p,v,event
`hyperleave2'		after leaving hyperlink	c,p,v,event
`iconclick1'	yes	before single click in icon box	c,p,v,event
`iconclick2'		after single click in icon box	c,p,v,event
`iconrclick1'	yes	before right click in icon box	c,p,v,event
`iconrclick2'		after right click in icon box	c,p,v,event
`icondclick1'	yes	before double click in icon box	c,p,v,event
`icondclick2'		after double click in icon box	c,p,v,event
`idle'		periodically (at idle time)	c
`init-color-markup'		(note 7)	colorer,p,v (note 7)
`menu1'	yes	before creating menus	c,p,v (note 3)
`menu2'	yes	during creating menus	c,p,v (note 3)
`menu-update'	yes	before updating menus	c,p,v
`new'		start of New command	c,old_c,new_c (note 9)
`open1'	yes	before opening any file	c,old_c,new_c,fileName (note 4)
`open2'		after opening any file	c,old_c,new_c,fileName (note 4)
`openwith1'	yes	before Open With command	c,p,v,d (note 14)
`openwith2'		after Open With command	c,p,v,(note 14)
`recentfiles1'	yes	before Recent Files command	c,p,v,fileName,closeFlag
`recentfiles2'		after Recent Files command	c,p,v,fileName,closeFlag
`redraw-entire-outline'	yes	start of tree.redraw	c (note 6)
`save1'	yes	before any Save command	c,p,v,fileName
`save2'		after any Save command	c,p,v,fileName
`scan-directives'		in scanDirectives	c,p,v,s,old_dict,dict,pluginsList (note 10)
`select1'	yes	before selecting a position	c,new_p,old_p,new_v,new_v
`select2'		after selecting a position	c,new_p,old_p,new_v,old_v
`select3'		after selecting a position	c,new_p,old_p,new_v,old_v
`set-mark'		when a mark is set	c,p,v
`show-popup-menu'		in tree.OnPopup	c,p,v,event
`start1'		after app.finishCreate()	None
`start2'		after opening first Leo window	c,p,v,fileName
`unselect1'	yes	before unselecting a vnode	c,new_p,old_p,new_v,old_v
`unselect2'		after unselecting a vnode	c,new_p,old_p,old_v,old_v
`@url1'	yes	before double-click @url node	c,p,v,url (note 5)
`@url2'		after double-click @url node	c,p,v(note 5)

Notes:

1. `activate' and `deactivate' hooks have been removed because they do not work as expected.
2. `commands' hooks: The label entry in the keywords dict contains the `canonicalized' form of the command, that is, the lowercase name of the command with all non-alphabetic characters removed. Commands hooks now set the label for undo and redo commands `undo' and `redo' rather than `cantundo' and `cantredo'.
3. `menu1' hook: Setting g.app.realMenuNameDict in this hook is an easy way of translating menu names to other languages. **Note**: the `new' names created this way affect only the actual spelling of the menu items, they do *not* affect how you specify shortcuts settings, nor do they affect the `official' command names passed in g.app.commandName. For example:

```
app().realMenuNameDict['Open...'] = 'Ouvre'.
```

4. `open1' and `open2' hooks: These are called with a keywords dict containing the following entries:
   - c: The commander of the newly opened window.
   - old_c: The commander of the previously open window.
   - new_c: (deprecated: use `c' instead) The commander of the newly opened window.
   - fileName: The name of the file being opened.

   You can use old_c.p and c.p to get the current position in the old and new windows. Leo calls the `open1' and `open2' hooks only if the file is not already open. Leo will also call the `open1' and `open2' hooks if: a) a file is opened using the Recent Files menu and b) the file is not already open.
5. `@url1' and `@url2' hooks are only executed if the `icondclick1' hook returns None.
6. These hooks are useful for testing.
7. These hooks allow plugins to parse and handle markup within doc parts, comments and Python `'' strings. Note that these hooks are *not* called in Python `'' strings. See the color_markup plugin for a complete example of how to use these hooks.
8. Leo calls the `create-optional-menus' hook when creating menus. This hook need only create new menus in the correct order, without worrying about the placement of the menus in the menu bar. See the plugins_menu and scripts_menu plugins for examples of how to use this hook.
9. The New command calls `new'. The `new_c' key is deprecated. Use the `c' key instead.
10. g.scanDirectives calls `scan-directives' hook. g.scanDirectives returns a dictionary, say d. d.get(`pluginsList') is an a list of tuples (d,v,s,k) where:
    - d is the spelling of the @directive, without the leading @.
    - v is the vnode containing the directive, _not_ the original vnode.
    - s[k:] is a string containing whatever follows the @directive. k has already been moved past any whitespace that follows the @directive.

    See the add_directives plugins directive for a complete example of how to use the `scan-directives' hook.
11. g.app.closeLeoWindow calls the `close-frame' hook just before removing the window from g.app.windowList. The hook code may remove the window from app.windowList to prevent g.app.closeLeoWindow from destroying the window.

12. Leo calls the `headkey1' and `headkey2' when the headline *might* have changed.
13. p is the new node (position) containing `@auto filename.ext'
14. New in Leo 4.10: the d argument to the open-with event handlers is a python dictionary whose keys are all the tags specified by the user in the body of the @open-with node.

12.2.1 Enabling idle time event handlers

Two methods in leoGlobals.py allow scripts and plugins to enable and disable `idle' events. **g.enableIdleTimeHook (idleTimeDelay=100)** enables the ``idle'' hook. Afterwards, Leo will call the ``idle'' hook approximately every idleTimeDelay milliseconds. Leo will continue to call the ``idle'' hook periodically until disableIdleTimeHook is called. **g.disableIdleTimeHook()** disables the ``idle'' hook.

12.3 Other topics

12.3.1 g.app.windowList: the list of all open frames

The windowlist attribute of the application instance contains the list of the frames of all open windows. The commands ivar of the frame gives the commander for that frame:

```
aList = g.app.windowList # get the list of all open frames.
g.es("windows...")
for f in aList:
    c = f.c # c is f's commander
    g.es(f)
    g.es(f.shortFileName())
    g.es(c)
    g.es(c.rootPosition())
```

There is also g.app.commanders() method, that gives the list of all active commanders directly.

12.3.2 Ensuring that positions are valid

Positions become invalid whenever the outline changes. Plugins and scripts that can make sure the position p is still valid by calling c.positionExists(p).

The following code will find a position p2 having the same vnode as p:

```
if not c.positionExists(p):
    for p2 in c.all_positions():
        if p2.v == p.v: # found
            c.selectPosition(p2)
    else:
        print('position no longer exists')
```

12.3.3 g.openWithFileName

g.openWithFileName opens a .leo file. For example:

```
new_c = g.openWithFileName(fileName,c)
```

new_c is the commander of the newly-created outline.

12.3.4 g.getScript

g.getScript(c,p) returns the expansion of p's body text. (If p is the presently selected node and there is a text selection, g.getScript returns the expansion of only the selected text.)

Leo scripts can use g.getScript to implement new ways of executing Python code. For example, the mod_scripting plugin uses g.getScript to implement @button nodes, and Leo's core uses g.getScript to implement @test nodes.

12.3.5 c.frame.body.bodyCtrl

Let:

```
w = c.frame.body.bodyCtrl # Leo's body pane.
```

Scripts can get or change the context of the body as follows:

```
w.appendText(s)                     # Append s to end of body text.
w.delete(i,j=None)                  # Delete characters from i to j.
w.deleteTextSelection()             # Delete the selected text, if any.
s = w.get(i,j=None)                 # Return the text from i to j.
s = w.getAllText                    # Return the entire body text.
i = w.getInsertPoint()              # Return the location of the cursor.
s = w.getSelectedText()             # Return the selected text, if any.
i,j = w.getSelectionRange (sort=True) # Return the range of selected text.
w.replace(i,j,s)                    # Replace the text from i to j by s.
w.setAllText(s)                     # Set the entire body text to s.
w.setSelectionRange(i,j,insert=None) # Select the text.
```

Notes:

- These are only the most commonly-used methods. For more information, consult Leo's source code.
- i and j are zero-based indices into the the text. When j is not specified, it defaults to i. When the sort parameter is in effect, getSelectionRange ensures i <= j.
- color is a Tk color name, even when using the Gt gui.

12.3.6 Invoking commands from scripts

Leo dispatches commands using c.doCommand, which calls the ``command1" and ``command2" hook routines for the given label. c.doCommand catches all exceptions thrown by the command:

```
c.doCommand(c.markHeadline,label="markheadline")
```

You can also call command handlers directly so that hooks will not be called:

```
c.markHeadline()
```

You can invoke minibuffer commands by name. For example:

```
c.executeMinibufferCommand('open-outline')
```

c.keyHandler.funcReturn contains the value returned from the command. In many cases, as above, this value is simply `break'.

12.3.7 Getting settings from @settings trees

Any .leo file may contain an @settings tree, so settings may be different for each commander. Plugins and other scripts can get the value of settings as follows:

```
format_headlines = c.config.getBool('rst3_format_headlines')
print('format_headlines',format_headlines)
```

The c.config class has the following getters. See the configSettings in leoCommands.py for details:

```
c.config.getBool(settingName,default=None)
c.config.getColor(settingName)
c.config.getDirectory(settingName)
c.config.getFloat(settingName)
c.config.getInt(settingName)
c.config.getLanguage(settingName)
c.config.getRatio(settingName)
c.config.getShortcut(settingName)
c.config.getString(settingName)
```

These methods return None if no setting exists. The getBool `default' argument to getBool gives the value to be returned if the setting does not exist.

12.3.8 Preferences ivars

Each commander maintains its own preferences. Your scripts can get the following ivars:

```
ivars = (
    'output_doc_flag',
    'page_width',
    'page_width',
    'tab_width',
    'target_language',
    'use_header_flag',
)

print("Prefs ivars...\n",color="purple")
```

```
for ivar in ivars:
    print(getattr(c,ivar))
```

If your script sets c.tab_width your script may call f.setTabWidth to redraw the screen:

```
c.tab_width = -4    # Change this and see what happens.
c.frame.setTabWidth(c.tab_width)
```

12.3.9 Functions defined in leoGlobals.py

leoGlobals.py contains many utility functions and constants. The following script prints all the names defined in leoGlobals.py:

```
print("Names defined in leoGlobals.py",color="purple")
names = g.__dict__.keys()
names.sort()
for name in names:
    print(name)
```

12.3.10 Making operations undoable

Plugins and scripts should call u.beforeX and u.afterX methods ato describe the operation that is being performed. **Note**: u is shorthand for c.undoer. Most u.beforeX methods return undoData that the client code merely passes to the corresponding u.afterX method. This data contains the `before' snapshot. The u.afterX methods then create a bead containing both the `before' and `after' snapshots.

u.beforeChangeGroup and u.afterChangeGroup allow multiple calls to u.beforeX and u.afterX methods to be treated as a single undoable entry. See the code for the Change All, Sort, Promote and Demote commands for examples. The u.beforeChangeGroup and u.afterChangeGroup methods substantially reduce the number of u.beforeX and afterX methods needed.

Plugins and scripts may define their own u.beforeX and afterX methods. Indeed, u.afterX merely needs to set the bunch.undoHelper and bunch.redoHelper ivars to the methods used to undo and redo the operation. See the code for the various u.beforeX and afterX methods for guidance.

p.setDirty and p.setAllAncestorAtFileNodesDirty now return a dirtyVnodeList that all vnodes that became dirty as the result of an operation. More than one list may be generated: client code is responsible for merging lists using the pattern dirtyVnodeList.extend(dirtyVnodeList2)

See the section << How Leo implements unlimited undo >> in leoUndo.py for more details. In general, the best way to see how to implement undo is to see how Leo's core calls the u.beforeX and afterX methods.

12.3.11 Redirecting output from scripts

leoGlobals.py defines 6 convenience methods for redirecting stdout and stderr:

```
g.redirectStderr() # Redirect stderr to the current log pane.
g.redirectStdout() # Redirect stdout to the current log pane.
```

```
g.restoreStderr()  # Restores stderr so it prints to the console window.
g.restoreStdout()  # Restores stdout so it prints to the console window.
g.stdErrIsRedirected() # Returns True if the stderr stream is redirected to the log pane.
g.stdOutIsRedirected() # Returns True if the stdout stream is redirected to the log pane.
```

Calls need *not* be paired. Redundant calls are ignored and the last call made controls where output for each stream goes. **Note**: you must execute Leo in a console window to see non-redirected output from the print statement:

```
print("stdout isRedirected: %s" % g.stdOutIsRedirected())
print("stderr isRedirected: %s" % g.stdErrIsRedirected())

g.redirectStderr()
print("stdout isRedirected: %s" % g.stdOutIsRedirected())
print("stderr isRedirected: %s" % g.stdErrIsRedirected())

g.redirectStdout()
print("stdout isRedirected: %s" % g.stdOutIsRedirected())
print("stderr isRedirected: %s" % g.stdErrIsRedirected())

g.restoreStderr()
print("stdout isRedirected: %s" % g.stdOutIsRedirected())
print("stderr isRedirected: %s" % g.stdErrIsRedirected())

g.restoreStdout()
print("stdout isRedirected: %s" % g.stdOutIsRedirected())
print("stderr isRedirected: %s" % g.stdErrIsRedirected())
```

12.3.12 Creating Qt Windows from Leo scripts

The following puts up a test window when run as a Leo script:

```
from PyQt4 import QtGui
w = QtGui.QWidget()
w.resize(250, 150)
w.move(300, 300)
w.setWindowTitle('Simple test')
w.show()
c.my_test = w # <-- Keep a reference to the window!
```

Important: Something like the last line is essential. Without it, the window would immediately disappear after being created. The assignment:

```
c.my_test = w
```

creates a permanent reference to the window so the window won't be garbage collected after the Leo script exits.

12.3.13 Writing to different log tabs

Plugins and scripts can create new tabs in the log panel. The following creates a tab named test or make it visible if it already exists:

```
c.frame.log.selectTab('Test')
```

g.es, g.enl, g.ecnl, g.ecnls write to the log tab specified by the optional tabName argument. The default for tabName is `Log'. The put and putnl methods of the gui's log class also take an optional tabName argument which defaults to `Log'.

Plugins and scripts may call the c.frame.canvas.createCanvas method to create a log tab containing a graphics widget. Here is an example script:

```
log = c.frame.log ; tag = 'my-canvas'
w = log.canvasDict.get(tag)
if not w:
    w = log.createCanvas(tag)
    w.configure(bg='yellow')
log.selectTab(tag)
```

12.3.14 Invoking dialogs using the g.app.gui class

Scripts can invoke various dialogs using the following methods of the g.app.gui object. Here is a partial list. You can use typing completion(default bindings: Alt-1 and Alt-2) to get the full list!

```
g.app.gui.runAskOkCancelNumberDialog(c,title,message)
g.app.gui.runAskOkCancelStringDialog(c,title,message)
g.app.gui.runAskOkDialog(c,title,message=None,text='Ok')
g.app.gui.runAskYesNoCancelDialog(c,title,message=None,
    yesMessage='Yes',noMessage='No',defaultButton='Yes')
g.app.gui.runAskYesNoDialog(c,title,message=None)
```

The values returned are in (`ok','yes','no','cancel'), as indicated by the method names. Some dialogs also return strings or numbers, again as indicated by their names.

Scripts can run File Open and Save dialogs with these methods:

```
g.app.gui.runOpenFileDialog(title,filetypes,defaultextension,multiple=False)
g.app.gui.runSaveFileDialog(initialfile,title,filetypes,defaultextension)
```

For details about how to use these file dialogs, look for examples in Leo's own source code. The runOpenFileDialog returns a list of file names.

12.3.15 Inserting and deleting icons

You can add an icon to the presently selected node with c.editCommands.insertIconFromFile(path). path is an absolute path or a path relative to the leo/Icons folder. A relative path is recommended if you plan to use the icons on machines with different directory structures.

For example:

```
path = 'rt_arrow_disabled.gif'
c.editCommands.insertIconFromFile(path)
```

Scripts can delete icons from the presently selected node using the following methods:

```
c.editCommands.deleteFirstIcon()
c.editCommands.deleteLastIcon()
c.editCommands.deleteNodeIcons()
```

12.3.16 Working with directives and paths

Scripts can easily determine what directives are in effect at a particular position in an outline. c.scanAllDirectives(p) returns a Python dictionary whose keys are directive names and whose values are the value in effect at position p. For example:

```
d = c.scanAllDirectives(p)
g.es(g.dictToString(d))
```

In particular, d.get(`path') returns the full, absolute path created by all @path directives that are in ancestors of node p. If p is any kind of @file node (including @file, @auto, @nosent, @shadow, etc.), the following script will print the full path to the created file:

```
path = d.get('path')
name = p.anyAtFileNodeName()
if name:
   name = g.os_path_finalize_join(path,name)
   g.es(name)
```

12.3.17 Running Leo in batch mode

On startup, Leo looks for two arguments of the form:

```
--script scriptFile
```

If found, Leo enters batch mode. In batch mode Leo does not show any windows. Leo assumes the scriptFile contains a Python script and executes the contents of that file using Leo's Execute Script command. By default, Leo sends all output to the console window. Scripts in the scriptFile may disable or enable this output by calling app.log.disable or app.log.enable

Scripts in the scriptFile may execute any of Leo's commands except the Edit Body and Edit Headline commands. Those commands require interaction with the user. For example, the following batch script reads a Leo file and prints all the headlines in that file:

```
path = g.os_path_finalize_join(g.app.loadDir,'..','test','test.leo')
assert g.os_path_exists(path),path

g.app.log.disable() # disable reading messages while opening the file
c2 = g.openWithFileName(path)
g.app.log.enable() # re-enable the log.
```

```
for p in c2.all_positions():
    g.es(g.toEncodedString(p.h,"utf-8"))
```

12.3.18 Getting interactive input from scripts

The following code can be run from a script to get input from the user using the minibuffer:

```
def getInput (event=None):

   stateName = 'get-input'
   k = c.k
   state = k.getState(stateName)

   if state == 0:
       k.setLabelBlue('Input: ',protect=True)
       k.getArg(event,stateName,1,getInput)
   else:
       k.clearState()
       g.es_print('input: %s' % k.arg)

getInput()
```

Let's look at this in detail. The lines:

```
stateName = 'get-input'
k = c.k
state = k.getState(stateName)
```

define a state *name*, `get-input', unique to this code. k.getState returns the present state (an int) associated with this state.

When getInput() is first called, the state returned by k.getState will be 0, so the following lines are executed:

```
if state == 0:
    k.setLabelBlue('Input: ',protect=True)
    k.getArg(event,stateName,1,getInput)
```

These lines put a protected label in the minibuffer: the user can't delete the label by backspacing. getArg, and the rest of Leo's key handling code, take care of the extremely complex details of handling key strokes in states. The call to getArg never returns. Instead, when the user has finished entering the input by typing <Return> getArg calls getInput so that k.getState will return state 1, the value passed as the third argument to k.getArg. The following lines handle state 1:

```
else:
    k.clearState()
    g.es_print('input: %s' % k.arg)
```

k.arg is the value returned by k.getArg. This example code just prints the value of k.arg and clears the input state.

12.3.19 The @g.command decorator

You can use the @g.command decorator to create new commands. This is an easy-to-use wrapper for c.k.registerCommand (), with the following advantages over it:

- The new command is automatically created for all Leo controllers (open Leo documents).
- The new command is also automatically available on all new Leo controllers (documents that will be opened in the future).
- Prettier syntax.

Therefore, @g.command can be naturally prototyped with execute-script (Ctrl+b) in Leo node.

As an example, you can execute this script to make command hello available:

```
@g.command('hello')
def hello_f(event):
    # use even['c'] to access controller
    c = event['c']
    pos = c.currentPosition()
    g.es('hello from', pos.h)
```

If you want to create a plugin that only exposes new commands, this is basically all you need in the plugins .py file. There is no need to hook up for `after-create-leo-frame' just to make your commands available.

If you want to create a command in object oriented style (so that the commands deal with your own objects), create them using closures like this (note how self is available inside command functions):

```
class MyCommands:
    def create(self):
        @g.command('foo1')
        def foo1_f(event):
            self.foo = 1

        @g.command('foo2')
        def foo2_f(event):
            self.foo = 2

        @g.command('foo-print')
        def foo_print_f(event):
            g.es('foo is', self.foo)

o = MyCommands()
o.create()
```

Note that running create() in this example in *after-create-leo-frame* is pointless - the newly created commands will override the commands in all previous controllers. You should consider this in your plugin design, and create your commands only once per Leo session.

12.3.20 Modifying plugins with @script scripts

The mod_scripting plugin runs @scripts before plugin initiation is complete. Thus, such scripts can not directly modify plugins. Instead, a script can create an event handler for the after-create-leo-frame that will modify the plugin.

For example, the following modifies the cleo.py plugin after Leo has completed loading it:

```
def prikey(self, v):
    try:
        pa = int(self.getat(v, 'priority'))
    except ValueError:
        pa = -1

    if pa == 24:
        pa = -1
    if pa == 10:
        pa = -2

    return pa

import types
from leo.core import leoPlugins

def on_create(tag, keywords):
    c.cleo.prikey = types.MethodType(prikey, c.cleo, c.cleo.__class__)

leoPlugins.registerHandler("after-create-leo-frame",on_create)
```

Attempting to modify c.cleo.prikey immediately in the @script gives an AttributeError as c has no .cleo when the @script is executed. Deferring it by using registerHandler() avoids the problem.

12.3.21 Creating minimal outlines

The following script will create a minimal Leo outline:

```
if 1:
    # Create a visible frame.
    c2 = g.app.newCommander(fileName=None)
else:
    # Create an invisible frame.
    c2 = g.app.newCommander(fileName=None,gui=g.app.nullGui)

c2.frame.createFirstTreeNode()
c2.redraw()

# Test that the script works.
for p in c2.all_positions():
    g.es(p.h)
```

12.3.22 Retaining pointers to Qt windows

The following script won't work as intended:

> from PyQt4 import QtGui w = QtGui.QWidget() w.resize(250, 150) w.move(300, 300) w.setWindowTitle (`Simple test') w.show()

When the script exits the sole reference to the window, w, ceases to exist, so the window is destroyed (garbage collected). To keep the window open, add the following code as the last line to keep the reference alive:

```
g.app.scriptsDict['my-script_w'] = w
```

Note that this reference will persist until the next time you run the execute-script. If you want something even more permanent, you can do something like:

```
g.app.my_script_w = w
```

第 **13** 章

Plugins

This chapter discusses the plugins contained in leoPlugins.leo. These plugins are part of Leo's official distribution. The next chapter, Writing Plugins, tells how to write plugins.

The scripting plugin (mod_scripting.py) deserves special mention. This plugin lets you create **script buttons** in a matter of seconds. See Creating script buttons. Script buttons are extraordinarily useful. Try them, you'll be instantly hooked.

Contents

- Plugins
  - Enabling plugins
  - Summary
  - Gui-independent plugins
    - Commands & directives
      - add_directives.py
      - bzr_qcommands.py
      - empty_leo_file.py
      - import_cisco_config.py
      - initinclass.py
      - leo_interface.py
      - lineNumbers.py
      - macros.py
      - mod_autosave.py
      - mod_read_dir_outline.py
      - mod_timestamp.py
      - nodeActions.py
      - outline_export.py
      - paste_as_headlines.py
      - pretty_print.py
      - quickMove.py
      - setHomeDirectory.py
      - word_count.py
    - Debugging
      - debugger_pudb.py
      - dump_globals.py
      - enable_gc.py
      - quit_leo.py
      - trace_gc_plugin.py
      - trace_keys.py
      - trace_tags.py
    - External programs
      - ipython.py
      - mod_tempfname.py
      - open_shell.py
      - tomboy_import.py
      - vim.py
      - xemacs.py
      - word_export.py
    - Files and nodes
      - active_path.py
      - at_folder.py
      - at_produce.py
      - at_view.py
      - backlink.py
      - datenodes.py
      - expfolder.py
      - FileActions.py

13.1 Enabling plugins

You enable or disable plugins using @enabled-plugins nodes in leoSettings files (leoSettings.leo, myLeoSettings.leo or the .leo file being loaded). See Specifying settings for full details of settings files.

The body text of the @enabled-plugins node contains a list of enabled plugins. Notes:

- **Leo attempts to load all plugins every time an @enabled-plugins node is seen.** If the plugin has already been loaded, Leo silently ignores the request to re-enable the plugin. Leo never attempts to disable a plugin while processing enabled plugin strings. Thus, plugins enabled in an @enabled-plugins node in leoSettings.leo *will* be enabled regardless of the contents of any other @enabled-plugins node.
- g.app.gui.getEnabledPlugins contains the last processed @enabled-plugins node.

13.2 Summary

active_path.py Synchronizes @path nodes with folders.

add_directives.py Allows users to define new @directives.

at_folder.py Synchronizes @folder nodes with folders.

at_produce.py Executes commands in nodes whose body text starts with @produce.

at_view.py Adds support for @clip, @view and @strip nodes.

attrib_edit.py Edits user attributes in a Qt frame.

backlink.py Allows arbitrary links between nodes.

bibtex.py Manages BibTeX files with Leo.

bzr_qcommands.py Adds a context menu to each node containing all the commands in the bzr Qt interface. Bzr is invoked based on the path of the current node.

chapter_hoist.py Creates hoist buttons.

colorize_headlines.py Manipulates appearance of individual tree widget items.

contextmenu.py Defines various useful actions for context menus (Qt only).

datenodes.py Allows users to insert headlines containing dates.

debugger_pudb.py Makes g.pdb() enter the Pudb debugger instead of pdb.

detect_urls.py Colorizes URLs everywhere in a node's body on node selection or saving. Double click on any URL launches it in the default browser.

dtest.py Sends code to the doctest module and reports the result.

dump_globals.py Dumps Python globals at startup.

EditAttributes.py Lets the user associate text with a specific node.

empty_leo_file.py Allows Leo to open any empty file as a minimal .leo file.

enable_gc.py Enables debugging and tracing for Python's garbage collector.

expfolder.py Adds @expfolder nodes that represent folders in the file system.

FileActions.py Defines actions taken when double-clicking on @<file> nodes and supports @file-ref nodes.

geotag.py Tags nodes with latitude and longitude.

graphcanvas.py Adds a graph layout for nodes in a tab. Requires Qt and the backlink.py plugin.

import_cisco_config.py Allows the user to import Cisco configuration files.

initinclass.py Modifies the Python @auto importer so that the importer puts the __init__ method (ctor) into the body of the class node.

interact.py Adds buttons so Leo can interact with command line environments.

ipython.py Creates a two-way communication (bridge) between Leo scripts and IPython running in the console from which Leo was launched.

leo_interface.py Allows the user to browse XML documents in Leo.

leo_pdf.py This NOT a Leo plugin: this is a docutils writer for .pdf files.

leo_to_html.py Converts a leo outline to an html web page.**.

leo_to_rtf.py Outputs a Leo outline as a numbered list to an RTF file. The RTF file can be loaded into Microsoft Word and formatted as a proper outline.

leocursor.py Creates a LeoCursor object that can walk around a Leo outline and decode attributes from nodes.

leoremote.py Remote control for Leo.

leoscreen.py Allows interaction with shell apps via screen.

lineNumbers.py Adds #line directives in perl and perlpod programs.

macros.py Creates new nodes containing parameterized section references.

maximizeNewWindows.py Maximizes all new windows.

mime.py Opens files with their default platform program.

mod_autosave.py Autosaves the Leo outline every so often.

mod_framesize.py Sets a hard coded frame size.

mod_http.py A minimal http plugin for LEO, based on AsyncHttpServer.py.

mod_read_dir_outline.py Allows Leo to read a complete directory tree into a Leo outline. Converts directories into headlines and puts the list of file names into bodies.

mod_scripting.py Creates script buttons and @button, @command, @plugin and @script nodes.

mod_tempfname.py Replaces c.openWithTempFilePath to create alternate temporary directory paths.

mod_timestamp.py Timestamps all save operations to show when they occur.

multifile.py Allows Leo to write a file to multiple locations.

nav_qt.py Adds ``Back'' and ``Forward'' buttons (Qt only).

niceNosent.py Ensures that all descendants of @file-nosent nodes end with exactly one newline, replaces all tabs with spaces, and adds a newline before class and functions in the derived file.

nodeActions.py Allows the definition of double-click actions.

open_shell.py Creates an `Extensions' menu containing two commands: Open Console Window and Open Explorer.

outline_export.py Modifies the way exported outlines are written.

paste_as_headlines.py Creates new headlines from clipboard text.

plugins_menu.py Creates a Plugins menu and adds all actives plugins to it.

pretty_print.py Customizes pretty printing.

projectwizard.py Creates a wizard that creates @auto nodes.

quickMove.py Creates buttons to move nodes quickly to other nodes.

quicksearch.py Adds a fast-to-use search widget, like the ``Find in files" feature of many editors.

quit_leo.py Shows how to force Leo to quit.

read_only_nodes.py Creates and updates @read-only nodes.

redirect_to_log.py Sends all output to the log pane.

run_nodes.py Runs a program and interface Leos through its input/output/error streams.

screenshots.py Creates stand-alone slideshows containing screenshots.

script_io_to_body.py Sends output from the Execute Script command to the end of the body pane.

scripts_menu.py Creates a Scripts menu for LeoPy.leo.

scrolledmessage.py Provides a Scrolled Message Dialog service for Qt.

setHomeDirectory.py Sets g.app.homeDir to a hard-coded path.

slideshow.py Support slideshows in Leo outlines.

spydershell.py Launches the spyder environment with access to Leo instance. See http://packages.python.org/spyder/.

startfile.py Launches (starts) a file given by a headline when double-clicking the icon.

stickynotes.py Adds simple ``sticky notes" feature (popout editors) for Qt gui.

todo.py Provides to-do list and simple task management for leo (Qt only).

tomboy_import.py Allows imports of notes created in Tomboy / gnote.

trace_gc_plugin.py Traces changes to Leo's objects at idle time.

trace_keys.py Traces keystrokes in the outline and body panes.

trace_tags.py Traces most common hooks, but not key, drag or idle hooks.

viewrendered.py Creates a window for *live* rendering of rst, html, etc. This plugin uses docutils, http:// docutils.sourceforge.net/, to do the rendering, so installing docutils is recommended.

vim.py Enables two-way communication with VIM.

word_count.py Counts characters, words, lines, and paragraphs in the body pane.

word_export.py Adds the Plugins:Word Export:Export menu item to format and export the selected outline to a Word document, starting Word if necessary.

xemacs.py Allows you to edit nodes in emacs/xemacs.

xsltWithNodes.py Adds the Outline:XSLT menu containing XSLT-related commands.

zenity_file_dialogs.py Replaces Leo's file dialogs on Linux with external calls to the zenity gtk dialog package.

13.3 Gui-independent plugins

13.3.1 Commands & directives

add_directives.py

Allows users to define new @direcives.

bzr_qcommands.py

Adds a context menu to each node containing all the commands in the bzr Qt interface. Bzr is invoked based on the path of the current node.

Requires contextmenu.py.

empty_leo_file.py

Allows Leo to open any empty file as a minimal .leo file.

import_cisco_config.py

Allows the user to import Cisco configuration files.

Adds the ``File:Import:Import Cisco Configuration'' menu item. The plugin will:

1. Create a new node, under the current node, where the configuration will be written. This node will typically have references to several sections (see below).
2. Create sections (child nodes) for the indented blocks present in the original config file. These child nodes will have sub-nodes grouping similar blocks (e.g. there will be an `interface' child node, with as many sub-nodes as there are real interfaces in the configuration file).
3. Create sections for the custom keywords specified in the customBlocks[] list in importCiscoConfig(). You can modify this list to specify different keywords. DO NOT put keywords that are followed by indented blocks (these are taken care of by point 2 above). The negated form of the keywords (for example, if the keyword is `service', the negated form is `no service') is also included in the sections.
4. Not display consecutive empty comment lines (lines with only a `!').

All created sections are alphabetically ordered.

initinclass.py

Modifies the Python @auto importer so that the importer puts the __init__ method (ctor) into the body of the class node.

This makes it easier to keep the instance variable docs in the class docstring in sync. with the ivars as manipulated by __init__, saves repeating explanations in both places.

Note that this is done *after* the consistency checks by the @auto import code, so using this plugin is at your own risk. It will change the order of declarations if other methods are declared before __init__.

leo_interface.py

Allows the user to browse XML documents in Leo.

This file implements an interface to XML generation, so that the resulting file can be processed by leo.

lineNumbers.py

Adds #line directives in perl and perlpod programs.

Over-rides two methods in leoAtFile.py to write #line directives after node sentinels. This allows compilers to give locations of errors in relation to the node name rather than the filename. Currently supports only perl and perlpod.

macros.py

Creates new nodes containing parameterized section reference.

This plugin adds nodes under the currently selected tree that are to act as section references. To do so, go the Outline menu and select the `Parameterize Section Reference' command. This plugin looks for a top level node called `Parameterized Nodes'. If it finds a headline that matches the section reference it adds a node/nodes to the current tree.

To see this in action, do the following:

0. **Important**: in the examples below, type << instead of < < and type >> instead of > >. Docstrings can not contain section references!
1. Create a node called `Parameterized Nodes', with a sub-node called < < Meow >>. The body of < < Meow > > should have the text:

```
I mmmm sooo happy I could  < < 1$  > >.
But I don't know if I have all the  < < 2$  > >
money in the world.
```

2. In a node called A, type:

```
< < meow( purrrrrr, zzooot )  > >
(leave the cursor at the end of the line)
```

3. In a node called B, type:

```
 < < meow ( spit or puke, blinkin  )  > >
(leave the cursor at the end of the line)
```

4. Leave the cursor in Node A at the designated point.
5. Go to Outline and select Parameterize Section Reference.

The plugin searches the outline, goes to level one and finds a Node with the Headline, ``Parameterized Nodes". It looks for nodes under that headline with the the headline << meow >>. It then creates this node structure under Node A:

```
< < meow ( purrrrrr, zzooot ) > >
  < <2$> >
  < <1$> >
```

6. Examine the new subnodes of Node A:

    < < meow (purrrrrr, zzooot) > > contains the body text of the < < meow > > node. < < 1$ > > contains the word purrrrrr. < < 2$ > > contains the word zzooot.

7. Go to Node B, and leave the cursor at the designated point.

Go to Outline Menu and select Parameterize Section Reference command.

8. Examine the new subnodes of Node B.

It's a lot easier to use than to explain!

mod_autosave.py

Autosaves the Leo outline every so often.

The time between saves is given by the setting, with default as shown:

```
@int mod_autosave_interval = 300
```

This plugin is active only if:

```
@bool mod_autosave_active = True
```

mod_read_dir_outline.py

Allows Leo to read a complete directory tree into a Leo outline. Converts directories into headlines and puts the list of file names into bodies.

Ce plug-in permet de traduire l'arborescence d'un rrtoire en une arborescence Leo : Chaque dossier est converti en noeud dans Leo ; son nom est placns l'ent du noeud et chaque nom de fichier qu'il contient est listns son contenu.

Feedback on this plugin can be sent to:

```
Frric Momm
<frederic [point] mommeja [at] laposte [point] net>
```

mod_timestamp.py

Timestamps all save operations to show when they occur.

nodeActions.py

Allows the definition of double-click actions.

When the user double-clicks a node this plugin checks for a match of the clicked node's headline text with a list of patterns. If a match occurs, the plugin executes the associated script.

nodeAction nodes may be located anywhere in the outline. Such nodes should contain one or more **pattern nodes** as children. The headline of each pattern node contains the pattern; the body text contains the script to be executed when the pattern matches the double-clicked node.

For example, the ``nodeActions'' node containing a ``launch URL'' pattern node and a ``pre-process python code'' node could be placed under an ``@settings'' node:

```
@settings
|
+- nodeActions
   |
   +- http:\\*
   |
   +- @file *.py
```

Configuration

The nodeActions plugin supports the following global configurations using Leo's support for setting global variables within an @settings node's sub-nodes in the leoSettings.leo, myLeoSettings.leo, and the project Leo file:

@bool nodeActions_save_atFile_nodes = False

True Double-click on an @file type node will save the file to disk before executing the script.

False Double-click on an @file type node will **not** save the file to disk before executing the script. (default)

@int nodeActions_message_level = 1

Specifies the type of messages to be sent to the log pane. Specifying a higher message level will display that level and all lower levels. The following integer values are supported:

```
0 no messages
1 Plugin triggered and the patterns that were matched (default)
2 Double-click event passed or not to next plugin
3 Patterns that did not match
4 Code debugging messages
```

Patterns

Pattern matching is performed using python's support for Unix shell-style patterns unless overwritten by the ``X'' pattern directive. The following pattern elements are supported:

```
*          matches everything
?          matches any single character
[<seq>]    matches any character in <seq>
[!<seq>]   matches any character **not** in <seq>
```

Unix shell-style pattern matching is case insensitive and always starts from the beginning of the headline. For example:

Pattern	Matches	Does not match
*.py	Abc_Test.py	
.py	.py - Test	Abc_Test.py
test*	Test_Abc.py	Abc_Test.py

To enable a script to run on any type of @file node (@thin, @shadow, ...), the pattern can start with ``@files" to match on any external file type. For example, the pattern ``@files *.py" will match a node with the headline ``@file abcd.py".

The headline of the double-clicked node is matched against the patterns starting from the first sub-node under the ``node-Actions" node to the last sub-node.

Only the script associated with the first matching pattern is invoked unless overwritten by the ``V" pattern directive.

Using the ``V" pattern directive allows a broad pattern such as ``@files *.py" to be invoked, and then, by placing a more restrictive pattern above it, such as ``@files *_test.py", a different script can be executed for those files requiring pre-processing:

```
+- nodeActions
  |
  +- @files *_test.py
  |
  +- @files *.py
```

Note: To prevent Leo from trying to save patterns that begin with a derived file directive (@file, @auto, ...) to disk, such as ``@file *.py", place the ``@ignore" directive in the body of the ``nodeActions" node.

Pattern nodes can be placed at any level under the ``nodeActions" node. Only nodes with no child nodes are considered pattern nodes. This allows patterns that are to be used in multiple Leo files to be read from a file. For example, the following structure reads the pattern definition from the ``C:\Leo\nodeActions_Patterns.txt" file:

```
+- nodeActions
|
+- @files C:\\Leo\\nodeActions_Patterns.txt
   |
   +- http:\\*
   |
   +- @file *.py
```

Pattern directives

The following pattern specific directives can be appended to the end of a pattern (do not include the `:'):

[X] Use python's regular expression type patterns instead of the Unix shell-style pattern syntax.

For example, the following patterns will match the same headline string:

```
Unix shell-style pattern:
  @files *.py

Regular Expression pattern:
  ^@files .*\.py$ [X]
```

[V] Matching the pattern will not block the double-click event from being passed to the remaining patterns. The ``V'' represents a down arrow that symbolizes the passing of the event to the next pattern below it.

For example, adding the ``[V]'' directive to the ``@files *_test.py'' in the Patterns section above, changes its script from being `an alternate to' to being `a pre-processor for' the ``@files *.py'' script:

```
+- nodeActions
  |
  +- @files *_test.py [V]
  |
  +- @files *.py
```

[>] Matching the pattern will not block the double-click event from being passed to other plugins. The ``>'' represents a right arrow that symbolizes the passing of the event to the next plugin.

If the headline matched more than one headline, the double-click event will be passed to the next plugin if the directive is associated with any of the matched patterns.

The directive(s) for a pattern must be contained within a single set of brackets, separated from the pattern by a space, with or without a comma separator. For example, the following specifies all three directives:

```
^@files .*\.py$ [X,V>]
```

Scripts

The script for a pattern is located in the body of the pattern's node. The following global variables are available to the script:

```
c
g
pClicked - node position of the double-clicked node
pScript - node position of the invoked script
```

Examples

Double-clicking on a node with a ``http:\\www.google.com'' headline will invoke the script associated with the ``http:\\*'' pattern. The following script in the body of the pattern's node displays the URL in a browser:

```
import webbrowser
hClicked = pClicked.h     #Clicked node's Headline text
webbrowser.open(hClicked) #Invoke browser
```

The following script can be placed in the body of a pattern's node to execute a command in the first line of the body of a double-clicked node:

```
g.os.system('"Start /b ' + pClicked.bodyString() + '"')
```

outline_export.py

Modifies the way exported outlines are written.

paste_as_headlines.py

Creates new headlines from clipboard text.

If the pasted text would be greater than 50 characters in length, the plugin truncates the headline to 50 characters and pastes the entire line into the body text of that node. Creates a ``Paste as Headlines" option the Edit menu directly under the existing Paste option.

pretty_print.py

Customizes pretty printing.

The plugin creates a do-nothing subclass of the default pretty printer. To customize, simply override in this file the methods of the base prettyPrinter class in leoCommands.py. You would typically want to override putNormalToken or its allies. Templates for these methods have been provided. You may, however, override any methods you like. You could even define your own class entirely, provided you implement the prettyPrintNode method.

quickMove.py

Creates buttons to move nodes quickly to other nodes.

Quickly move/copy/clone nodes from around the tree to one or more target nodes. It can also create bookmark and tagging functionality in an outline (see *Set Parent Notes* below).

Adds *Move/Clone/Copy To Last Child Button* and *Move/Clone/Copy To First Child Button*, *Link To/From* and *Jump To* commands to the Move sub-menu on the Outline menu, and each node's context menu, if the *contextmenu* plugin is enabled.

Select a node Foo and then use the *Move To Last Child Button* command. This adds a `to Foo' button to the button bar. Now select another node and click the `to Foo' button. The selected node will be moved to the last child of the node `Foo'.

To First Child Button works the same way, except that moved nodes are inserted as the first child of the target node.

Clone and *Copy* variants are like *Move*, but clone or copy instead of moving.

Link works in conjunction with the *backlink* plugin (and also the *graphcanvas* plugin) creating a link to/from the target and current nodes.

Jump buttons act as bookmarks, taking you to the target node.

You can right click on any of these buttons to access their context menu:

Goto Target takes you to the target node (like a *Jump* button).

Make Permanent makes the button permanent, it will reappear when the file is saved / closed / re-opened.

Set Parent allows you to move buttons to sub-menu items of other *quickMove* buttons. This implicitly makes the moved button permanent. It also causes the moved button to lose its context menu.

Remove Button comes from the *mod_scripting* plugin, and just removes the button for the rest of the current session.

Set Parent Notes *Set Parent* doesn't allow you to do anything with *quickMove* you couldn't do with a long strip of separate buttons, but it collects quickMove buttons as sub-menu items of one quickMove button, saving a lot of toolbar space.

Bookmarks Create somewhere out of the way in your outline a node called *Bookmarks*. Use the quickMove menu to make it a *Jump To* button, and use its context menu to make it permanent. There is no particular reason to jump to it, but it needs to be a *quickMove* button of some kind.

Now, when you want to bookmark a node, first use the quickMove menu to make the node a *Jump To* button, and then use the context menu on the button to set its parent to your *Bookmarks* button. It becomes a sub-menu item of the *Bookmarks* button.

Tags In conjunction with the *backlinks* plugin you can use *quickMove* to tag nodes. The *backlinks* plugin adds a *Links* tab to the *Log pane*.

Create somewhere in your outline a node called *Tags*. Use the quickMove menu to make it a *Jump To* button, and use its context menu to make it permanent. Clicking on it will jump you to your tag list. Now create a node under the *Tags* node for each tag you want. The node's name will be the tag name, and can be changed later. Then use the quickMove menu to make each of these nodes a *Link To* button, and then use the context menu on the button to set its parent to your *Tags* button. It becomes a sub-menu item of the *Tags* button.

To see the tags on a node, you need to be looking at the *Links* tab in the *Log pane*. To see all the nodes with a particular tag, click on the *Tags* button to jump to the tag list, and select the node which names the tag of interest. The nodes with that tag will be listed in th *Links* tab in the *Log pane*.

setHomeDirectory.py

Sets g.app.homeDir to a hard-coded path.

word_count.py

Counts characters, words, lines, and paragraphs in the body pane.

It adds a ``Word Count..." option to the bottom of the Edit menu that will activate the command.

13.3.2 Debugging

debugger_pudb.py

Makes g.pdb() enter the Pudb debugger instead of pdb.

Pudb is a full-screen Python debugger: http://pypi.python.org/pypi/pudb

dump_globals.py

Dumps Python globals at startup.

enable_gc.py

Enables debugging and tracing for Python's garbage collector.

quit_leo.py

Shows how to force Leo to quit.

trace_gc_plugin.py

Traces changes to Leo's objects at idle time.

trace_keys.py

Traces keystrokes in the outline and body panes.

trace_tags.py

Traces most common hooks, but not key, drag or idle hooks.

13.3.3 External programs

ipython.py

Creates a two-way communication (bridge) between Leo scripts and IPython running in the console from which Leo was launched.

Using this bridge, scripts running in Leo can affect IPython, and vice versa. In particular, scripts running in IPython can alter Leo outlines!

For full details, see Leo Users Guide: http://webpages.charter.net/edreamleo/IPythonBridge.html

mod_tempfname.py

Replaces c.openWithTempFilePath to create alternate temporary directory paths.

Two alternates are supported. The default method creates temporary files with a filename that begins with the headline text, and located in a ``username_Leo'' subdirectory of the temporary directory. The ``LeoTemp'' prefix is omitted. If `open_with_clean_filenames' is set to true then subdirectories mirror the node's hierarchy in Leo. Either method makes it easier to see which temporary file is related to which outline node.

open_shell.py

Creates an `Extensions' menu containing two commands: Open Console Window and Open Explorer.

The Open Console Window command opens xterm on Linux. The Open Explorer command Opens a Windows explorer window.

This allows quick navigation to facilitate testing and navigating large systems with complex directories.

Please submit bugs / feature requests to etaekema@earthlink.net

Current limitations: - Not tested on Mac OS X ... - On Linux, xterm must be in your path.

tomboy_import.py

Allows imports of notes created in Tomboy / gnote.

Usage:

- Create a node with the headline `tomboy'
- Select the node, and do alt+x act-on-node
- The notes will appear as children of `tomboy' node
- The next time you do act-on-node, existing notes will be updated (they don't need to be under `tomboy' node anymore) and new notes added.

vim.py

Enables two-way communication with VIM.

It's recommended that you have gvim installed--the basic console vim is not recommended.

When properly installed, this plugin does the following:

- By default, the plugin opens nodes on icondclick2 events. (double click in the icon box)
- The setting:

  ```
  @string vim_trigger_event = icondclick2
  ```

  controls when nodes are opened in vim. The default, shown above, opens a node in vim on double clicks in Leo's icon box. A typical alternative would be:

  ```
  @string vim_trigger_event = iconclick2
  ```

  to open nodes on single clicks in the icon box. You could also set:

  > @string vim_trigger_event = select2

  to open a node in vim whenever the selected node changes for any reason.
- Leo will put Vim cursor at same location as Leo cursor in file if `vim_plugin_positions_cursor' set to True.
- Leo will put node in a Vim tab card if `vim_plugin_uses_tab_feature' set to True.
- Leo will update the node in the outline when you save the file in VIM.

To install this plugin do the following:

1. On Windows, set the vim_cmd and vim_exe settings to the path to vim or gvim as shown in leoSettings.leo. Alternatively, you can ensure that gvim.exe is on your PATH.

1. If you are using Python 2.4 or above, that's all you need to do. Jim Sizelove's new code will start vim automatically using Python's subprocess module. The subprocess module comes standard with Python 2.4. For Linux systems, Leo will use subprocess.py in Leo's extensions folder if necessary.

xemacs.py

Allows you to edit nodes in emacs/xemacs.

Important: the open_with plugin must be enabled for this plugin to work properly.

Depending on your preference, selecting or double-clicking a node will pass the body text of that node to emacs. You may edit the node in the emacs buffer and changes will appear in Leo.

word_export.py

Adds the Plugins:Word Export:Export menu item to format and export the selected outline to a Word document, starting Word if necessary.

13.3.4 Files and nodes

active_path.py

Synchronizes @path nodes with folders.

If a node is named `@path path_to_folder', the content (file and folder names) of the folder and the children of that node will synchronized whenever the node's status-iconbox is double clicked.

For files not previously seen in a folder a new node will appear on top of the children list (with a mark).

Folders appear in the list as /foldername/. If you double click on the icon-box of the folder node, it will have children added to it based on the contents of the folder on disk. These folders have the `@path` directive as the first line of their body text.

When files are deleted from the folder and the list is updated by double clicking the files will appear in the list as *filename* (or */foldername/*).

You can describe files and directories in the body of the nodes.

You can organize files and directories with organizer nodes, an organizer node name cannot contain with `/'.

Files and folders can be created by entering a node with the required name as its headline (must start and/or end with ``/'' for a folder) and then double clicking on the node's status-iconbox.

@auto nodes can be set up for existing files can be loaded by double clicking on the node's status-iconbox. If you prefer @shadow or something else use the ``active_path_attype'' setting, without the ``@''.

There are commands on the Plugins active_path submenu:

- show path - show the current path
- set absolute path - changes a node ``/dirname/'' to ``@path /absolute/path/to/dirname''.
- purge vanished (recursive) - remove *entries*
- update recursive - recursive load of directories, use with caution on large file systems

If you want to use an input other than double clicking a node's status-iconbox set active_path_event to a value like `iconrclick1' or `iconclick1'.

There are @settings for ignoring directory entries and automatically loading files. re.search is used, rather than re.match, so patterns need only match part of the filename, not the whole filename.

The body of the @setting @data active_path_ignore is a list of regex patterns, one per line. Directory entries matching any pattern in the list will be ignored. The names of directories used for matching will have forward slashes around them (`/ dirname/'), so patterns can use this to distinguish between directories and files.

The body of the @setting @data active_path_autoload is a list of regex patterns, one per line. File entries matching any pattern in the list will be loaded automatically. This works only with files, not directories (but you can load directories recursively anyway).

Set @bool active_path_load_docstring = True to have active_path load the docstring of .py files automatically. These nodes start with the special string:

```
@language rest # AUTOLOADED DOCSTRING
```

which must be left intact if you want active path to be able to double-click load the file later.

@float active_path_timeout_seconds (default 10.) controls the maximum time active_path will spend on a recursive operation.

@int active_path_max_size (default 1000000) controls the maximum size file active_path will open without query.

active_path is a rewrite of the at_directory plugin to use @path directives (which influence @auto and other @file type directives), and to handle sub-folders more automatically.

at_folder.py

Synchronizes @folder nodes with folders.

If a node is named `@folder path_to_folder', the content (filenames) of the folder and the children of that node will be sync. Whenever a new file is put there, a new node will appear on top of the children list (with mark). So that I can put my description (i.e. annotation) as the content of that node. In this way, I can find any files much easier from leo.

Moreover, I add another feature to allow you to group files(in leo) into children of another group. This will help when there are many files in that folder. You can logically group it in leo (or even clone it to many groups), while keep every files in a flat/single directory on your computer.

at_produce.py

Executes commands in nodes whose body text starts with @produce.

To use, put in the body text of a node:

```
@produce javac -verbose Test.java
```

To execute, you goto Outline and look at Produce. Choose Execute All Produce or Execute Tree Produce. The Tree does the current Tree, All does the whole Outline. Executing will fire javac, or whatever your using. @produce functions as a directive. After executing, a log file/node is created at the top of the Outline. Any output, even error messages, should be there.

It executes in a hierarchal manner. Nodes that come before that contain @produce go first.

I'm hoping that this orthogonal to @run nodes and anything like that. Its not intended as a replacement for make or Ant, but as a simple substitute when that machinery is overkill.

WARNING: trying to execute a non-existent command will hang Leo.

at_view.py

Adds support for @clip, @view and @strip nodes.

- Selecting a headline containing @clip appends the contents of the clipboard to the end of the body pane.
- Double clicking the icon box of a node whose headline contains @view <*path-to-file*> places the contents of the file in the body pane.
- Double clicking the icon box of a node whose headline contains @strip <*path-to-file*> places the contents of the file in the body pane, with all sentinels removed.

This plugin also accumulates the effect of all @path nodes.

backlink.py

Allows arbitrary links between nodes.

datenodes.py

Allows users to insert headlines containing dates.

`Date nodes' are nodes that have dates in their headlines. They may be added to the outline one at a time, a month's-worth at a time, or a year's-worth at a time. The format of the labels (headlines) is configurable.

There are options to omit Saturdays and Sundays.

An `Insert Date Nodes ...' submenu will be created (by default) in the `Outline' menu. This menu can be suppressed by using either of the following settings:

- @bool suppress-datenodes-menus
- @bool suppress-all-plugins-menus

The following commands are available for use via the minibuffer or in @menu/@popup settings.

- datenodes-today
- datenodes-this-month
- datenodes-this-year

expfolder.py

Adds @expfolder nodes that represent folders in the file system.

Double clicking on the icon of an @expfolder heading reads the files in the directory at the path specified and creates child nodes for each file in the subfolder. Subdirectories are made into child @expfolder nodes so the tree can be easily traversed. If files have extensions specified in the expfolder.ini file they are made into @text nodes so the content of the files can be easily loaded into leo and edited. Double clicking a second time will delete all child nodes and refresh the directory listing. If there are any changed @text nodes contained inside you will be prompted about saving them.

The textextensions field on the expfolder Properties page contains a list of extensions which will be made into @text nodes, separated by spaces.

For the @text and @expfolder nodes to interact correctly, the textnode plugin must load before the expfolder plugin. This can be set using the Plugin Manager's Plugin Load Order pane.

FileActions.py

Defines actions taken when double-clicking on @<file> nodes and supports @file-ref nodes.

Double-clicking any kind of @<file> node writes out the file if changes have been made since the last save, and then runs a script on it, which is retrieved from the outline.

Scripts are located in a node whose headline is FileActions. This node can be anywhere in the outline. If there is more than one such node, the first one in outline order is used.

The children of that node are expected to contain a file pattern in the headline and the script to be executed in the body. The file name is matched against the patterns (which are Unix-style shell patterns), and the first matching node is selected. If the filename is a path, only the last item is matched.

Execution of the scripts is similar to the ``Execute Script" command in Leo. The main difference is that the namespace in which the scripts are run contains these elements:

- `c' and `g' and `p': as in the regular execute script command.
- `filename': the filename from the @file directive.
- **`shellScriptInWindow', a utility function that runs a shell script in an** external windows, thus permitting programs to be called that require user interaction

File actions are implemented for all kinds @<file> nodes. There is also a new node type @file-ref for referring to files purely for the purpose of file actions, Leo does not do anything with or to such files.

geotag.py

Tags nodes with latitude and longitude.

leocursor.py

Creates a LeoCursor object that can walk around a Leo outline and decode attributes from nodes.

Node names can be used through . (dot) notation so cursor.Data.Name._B for example returns the body text of the Name node which is a child of the Data node which is a child of the cursors current location.

See .../plugins/examples/leocursorexample.leo for application.

mime.py

Opens files with their default platform program.

Double-clicking @mime nodes will attempt to open the named file as if opened from a file manager. @path parent nodes are used to find the full filename path. Fore example:

```
@mime foodir/document.pdf
```

The string setting `mime_open_cmd' allows specifying a program to handle opening files:

```
@settings
    @string mime_open_cmd = see
    .. or ..
    @string mime_open_cmd = see %s
```

Where `%s' is replaced with the full pathname.

Note: This plugin terminates handling of the `icondclick1' event by returning True. If another plugin using this event (e.g. vim.py) is also enabled, the order in @enabled-plugins matters. For example: if vim.py is enabled before mime.py, double-clicking on an @mime node will both open the body text in [g]vim AND call the mime_open_cmd.

This plugin is complementary to the UNL.py plugin's @url nodes. Use @url for opening either URLs or Uniform Node Locators in ``*.leo'' files and use @mime nodes for opening files on the local file system. It also replaces the startfile.py plugin, where here the headline must start with @mime to activate this plugin.

For other sys.platform's, add an elif case to the section ``guess file association handler'' and either define a default _mime_open_cmd string, where ``%s'' will be replaced with the filename, or define a function taking the filename string as its only argument and set as open_func.

multifile.py

Allows Leo to write a file to multiple locations.

This plugin acts as a post-write mechanism, a file must be written to the file system for it to work. At this point it is not a replacement for @path or an absolute path, it works in tandem with them.

To use, place @multipath at the start of a line in the root node or an ancestor of the node. The format is (On Unix-like systems):

```
@multipath /machine/unit/:/machine/robot/:/machine/
```

New in version 0.6 of this plugin: the separator used above is `;' not `:', for example:

```
@multipath c:\prog\test;c:\prog\unittest
```

It will places copy of the written file in each of these directories.

There is an additional directive that simplifies common paths, it is called @multiprefix. By typing @multiprefix with a path following it, before a @multipath directive you set the beginning of the paths in the @multipath directive. For example:

```
#@multiprefix /leo #@multipath /plugins
```

or:

```
#@multiprefix /leo/
#@multipath plugins: fungus : drain
```

copies a file to /leo/plugins /leo/fungus /leo/drain.

Note I put # in front of the directives here because I don't want someone browsing this file to accidentally save multiple copies of this file to their system :))

The @multiprefix stays in effect for the entire tree until reset with another @multiprefix directive. @multipath is cumulative, in that for each @multipath in an ancestor a copy of the file is created. These directives must at the beginning of the line and by themselves.

niceNosent.py

Ensures that all descendants of @file-nosent nodes end with exactly one newline, replaces all tabs with spaces, and adds a newline before class and functions in the derived file.

read_only_nodes.py

Creates and updates @read-only nodes.

Here's my first attempt at customizing leo. I wanted to have the ability to import files in ``read-only" mode, that is, in a mode where files could only be read by leo (not tangled), and also kept in sync with the content on the drive.

The reason for this is for example that I have external programs that generate resource files. I want these files to be part of a leo outline, but I don't want leo to tangle or in any way modify them. At the same time, I want them to be up-to-date in the leo outline.

So I coded the directive plugin. It has the following characteristics:

- It reads the specified file and puts it into the node content.
- If the @read-only directive was in the leo outline already, and the file content on disk has changed from what is stored in the outline, it marks the node as changed and prints a ``changed" message to the log window; if, on the other hand, the file content has _not_ changed, the file is simply read and the node is not marked as changed.
- When you write a @read-only directive, the file content is added to the node immediately, i.e. as soon as you press Enter (no need to call a menu entry to import the content).
- If you want to refresh/update the content of the file, just edit the headline and press Enter. The file is reloaded, and if in the meantime it has changed, a ``change" message is sent to the log window.
- The body text of a @read-only file cannot be modified in leo.

The syntax to access files in @read-only via ftp/http is the following:

```
@read-only http://www.ietf.org/rfc/rfc0791.txt
@read-only ftp://ftp.someserver.org/filepath
```

If FTP authentication (username/password) is required, it can be specified as follows:

```
@read-only ftp://username:password@ftp.someserver.org/filepath
```

For more details, see the doc string for the class FTPurl.

Davide Salomoni

run_nodes.py

Runs a program and interface Leos through its input/output/error streams.

Double clicking the icon box whose headlines are @run `cmd args' will execute the command. There are several other features, including @arg and @input nodes.

The run_nodes.py plugin introduce two new nodes that transform leo into a terminal. It was mostly intended to run compilers and debuggers while having the possibility to send messages to the program.

Double clicking on the icon of an node whose headline is @run <command> <args> will launch <command> with the given arguments. It will also mark the node. # Terminates the argument list. @run # <comment> is also valid.

@in nodes are used to send input to the running process. Double clicking on the icon of an @in <message> node will append a ``n'' to <message> and write it to the program, no matter where the node is placed. If no @run node is active, nothing happens.

The body text of every child, in which the headlines do not begin with `@run` or `@in`, will be appended to <command>, allowing you to add an unlimited number of arguments to <command>.

The output of the program is written in the log pane (Error output in red). When the program exit the node is set unmarked and the return value is displayed... When the enter key is pressed in the body pane of an active @run node the content of it body pane is written to the program and then emptied ready for another line of input. If the node have @run nodes in its descendants, they will be launched successively. (Unless one returned an exit code other than 0, then it will stop there)

By Alexis Gendron Paquette. Please send comments to the Leo forums.

startfile.py

Launches (starts) a file given by a headline when double-clicking the icon.

This plugin ignores headlines starting with an `@'. Uses the @folder path if the headline is under an @folder headline. Otherwise the path is relative to the Leo file.

xsltWithNodes.py

Adds the Outline:XSLT menu containing XSLT-related commands.

This menu contains the following items:

- **Set StyleSheet Node:**
    - Selects the current node as the xsl stylesheet the plugin will use.
- **Process Node with Stylesheet Node:**
    - Processes the current node as an xml document, resolving section references and Leo directives.
    - Creates a sibling containing the results.

Requires 4Suite 1.0a3 or better, downloadable from http://4Suite.org.

13.3.5 Scripting

dyna_menu

The dyna_menu plugin is a remarkable body of work by `e'. This plugin creates a dyna_menu menu from which you can execute commands. You may download the latest version at: http://rclick.netfirms.com/dyna_menu.py.html

leoscreen.py

Allows interaction with shell apps via screen.

Analysis environments like SQL, R, scipy, ipython, etc. can be used by pasting sections of text from an editor (Leo) and a shell window. Results can be pasted back into the editor.

This plugin streamlines the process by communicating with screen, the shell multiplexer

Commands

leoscreen-run-text Send the text selected in Leo's body text to the shell app. Selects the next line for your convenience.

leoscreen-get-line Insert a line of the last result from the shell into Leo's body text at the current insert point. Lines are pulled one at a time starting from the end of the output. Can be used repeatedly to get the output you want into Leo.

leoscreen-get-all Insert all of the last result from the shell into Leo's body text at the current insert point.

leoscreen-get-note Insert all of the last result from the shell into a new child node of the current node.

leoscreen-show-all Show the output from the last result from the shell in a temporary read only window. **Important**: The output is not stored.

leoscreen-show-note Insert all of the last result from the shell into a new child node of the current node and display that node a a stickynote (requires stickynote plugin).

leoscreen-next Switch screen session to next window.

leoscreen-prev Switch screen session to preceding window.

leoscreen-other Switch screen session to last window displayed.

leoscreen-get-prefix Interactively get prefix for inserting text into body (#, --, //, etc/) Can also set using:

```
c.leo_screen.get_line_prefix = '#'
```

leoscreen-more-prompt Skip one less line at the end of output when fetching output into Leo. Adjusts lines skipped to avoid pulling in the applications prompt line.

leoscreen-less-prompt Skip one more line at the end of output when fetching output into Leo Adjusts lines skipped to avoid pulling in the applications prompt line.

Settings

leoscreen_prefix Prepended to output pulled in to Leo. The substring SPACE in this setting will be replaced with a space character, to allow for trailing spaces.

leoscreen_time_fmt time.strftime format for note type output headings.

Theory of operation

leoscreen creates a instance at c.leo_screen which has some methods which might be useful in @button and other Leo contexts.

Example SQL setup

In a Leo file full of interactive SQL analysis, I have:

```
@settings
  @string leoscreen_prefix = --SPACE
@button rollback
  import time
  c.leo_screen.run_text('ROLLBACK;  -- %s\n' % time.asctime())
@button commit
  import time
  cmd = 'COMMIT;  -- %s' % time.asctime()
  c.leo_screen.run_text(cmd)
  c.leo_screen.insert_line(cmd)
```

which creates a button to rollback messed up queries, another to commit (requiring additional action to supply the newline as a safeguard) and sets the prefix to ``-- '' for text pulled back from the SQL session into Leo.

Implementation note: screen behave's differently if screen -X is executed with the same stdout as the target screen, vs. a different stdout. Although stdout is ignored, Popen() needs to ensure it's not just inherited.

mod_scripting.py

Creates script buttons and @button, @command, @plugin and @script nodes.

This plugin puts buttons in the icon area. Depending on settings the plugin will create the `Run Script', the `Script Button' and the `Debug Script' buttons.

The `Run Script' button is simply another way of doing the Execute Script command: it executes the selected text of the presently selected node, or the entire text if no text is selected.

The `Script Button' button creates *another* button in the icon area every time you push it. The name of the button is the headline of the presently selected node. Hitting this *newly created* button executes the button's script.

For example, to run a script on any part of an outline do the following:

1. Select the node containing the script.
2. Press the scriptButton button. This will create a new button.
3. Select the node on which you want to run the script.
4. Push the *new* button.

That's all.

For every @button node, this plugin creates two new minibuffer commands: x and delete-x-button, where x is the `cleaned' name of the button. The `x' command is equivalent to pushing the script button.

You can specify **global buttons** in leoSettings.leo or myLeoSettings.leo by putting @button nodes as children of an @buttons node in an @settings trees. Such buttons are included in all open .leo (in a slightly different color). Actually, you can specify global buttons in any .leo file, but @buttons nodes affect all later opened .leo files so usually you would define global buttons in leoSettings.leo or myLeoSettings.leo.

The cleaned name of an @button node is the headline text of the button with:

- Leading @button or @command removed,
- @key and all following text removed,
- @args and all following text removed,
- all non-alphanumeric characters converted to a single `-` characters.

Thus, cleaning headline text converts it to a valid minibuffer command name.

You can delete a script button by right-clicking on it, or by executing the delete-x-button command.

The `Debug Script' button runs a script using an external debugger.

This plugin optionally scans for @button nodes, @command, @plugin nodes and @script nodes whenever a .leo file is opened.

- @button nodes create script buttons.
- @command nodes create minibuffer commands.
- @plugin nodes cause plugins to be loaded.
- @script nodes cause a script to be executed when opening a .leo file.

Such nodes may be security risks. This plugin scans for such nodes only if the corresponding atButtonNodes, atPluginNodes, and atScriptNodes constants are set to True in this plugin.

You can specify the following options in leoSettings.leo. See the node: @settings-->Plugins-->scripting plugin. Recommended defaults are shown:

```
@bool scripting-at-button-nodes = True
True: adds a button for every @button node.

@bool scripting-at-commands-nodes = True
True: define a minibuffer command for every @command node.

@bool scripting-at-plugin-nodes = False
True: dynamically loads plugins in @plugins nodes when a window is created.

@bool scripting-at-script-nodes = False
True: dynamically executes script in @script nodes when a window is created.
This is dangerous!

@bool scripting-create-debug-button = False
True: create Debug Script button.

@bool scripting-create-run-script-button = False
True: create Run Script button.
Note: The plugin creates the press-run-script-button regardless of this setting.
```

```
@bool scripting-create-script-button-button = True
True: create Script Button button in icon area.
Note: The plugin creates the press-script-button-button regardless of this setting.

@int scripting-max-button-size = 18
The maximum length of button names: longer names are truncated.
```

You can bind key shortcuts to @button and @command nodes as follows.

@button name @key=shortcut

> Binds the shortcut to the script in the script button. The button's name is `name', but you can see the full headline in the status line when you move the mouse over the button.

@command name @key=shortcut

> Creates a new minibuffer command and binds shortcut to it. As with @buffer nodes, the name of the command is the cleaned name of the headline.

This plugin is based on ideas from e's dynabutton plugin, quite possibly the most brilliant idea in Leo's history.

You can run the script with sys.argv initialized to string values using @args. For example:

@button test-args @args = a,b,c

will set sys.argv to [u'a',u'b',u'c']

script_io_to_body.py

Sends output from the Execute Script command to the end of the body pane.

13.3.6 Servers

leoremote.py

Remote control for Leo.

Example client:

```
from leo.external import lproto
import os


addr = open(os.path.expanduser('~/.leo/leoserv_sockname')).read()
print("will connect to",addr)
pc  = lproto.LProtoClient(addr)
pc.send("""
   g.es("hello world from remote")
   c = g.app.commanders()[0]
""")
```

```
# note how c persists between calls
pc.send("""c.k.simulateCommand('stickynote')""")
```

mod_http.py

A minimal http plugin for LEO, based on AsyncHttpServer.py.

Use this plugin is as follows:

1. Start Leo with the plugin enabled. You will see a purple message that says something like:

```
"http serving enabled on port 8080, version 0.91"
```

2. Start a web browser, and enter the following url: http://localhost:8080/

You will see a a ``top" level page containing one link for every open .leo file. Start clicking :-)

You can use the browser's refresh button to update the top-level view in the browser after you have opened or closed files.

To enable this plugin put this into your file:

```
@settings
  @bool http_active = True
  @int  port = 8080
  @string rst_http_attributename = 'rst_http_attribute'
```

Note: the browser_encoding constant (defined in the top node of this file) must match the character encoding used in the browser. If it does not, non-ascii characters will look strange.

13.3.7 Slideshows and screenshots

screenshots.py

Creates stand-alone slideshows containing screenshots.

This plugin defines five commands. The **apropos-slides** command prints this message to Leo's log pane. The **slide-show-info** command prints the settings in effect.

The **make-slide** and **make-slide-show** commands, collectively called **slide commands**, create collections of slides from **@slideshow** trees containing **@slide** nodes.

Slides may link to screenshots. The slide commands can generate screenshots from **@screenshot-tree** nodes, but this feature has proven to be clumsy and inflexible. It is usually more convenient to use screenshots taken with a program such as Wink. The **meld-slides** command creates references to externally-generated screenshots within @slide nodes.

@slide nodes may contain **@url nodes**. These @url nodes serve two purposes. First, they allow you to see various files (slides, initial screenshots, working files and final screenshots). Second, these @url nodes guide the meld script and the four commands defined by this plugin (see below). By inserting or deleting these @url nodes you (or your scripts) can customize how the commands (and meld) work. In effect, the @url nodes become per-slide settings.

Prerequisites

Inkscape (Required) An SVG editor: http://www.inkscape.org/ Allows the user to edit screenshots. Required to create final output (PNG) files.

PIL (Optional but highly recommended) The Python Imaging Library, http://www.pythonware.com/products/pil/

Wink (Optional) A program that creates slideshows and slides. http://www.debugmode.com/wink/

Summary

@slideshow <slideshow-name> Creates the folder: <sphinx_path>/slides/<slideshow-name>

@slide <ignored text> Creates slide-<slide-number>.html (in the sphinx _build directory). **Note**: the plugin skips any @slide nodes with empty body text.

@screenshot Specifies the contents of the screenshot.

Options are child nodes of @slideshow or @slide nodes that control the make-slide and make-slide-show commands. See the Options section below.

The make-slide and make-slide-show commands create the following @url nodes as children of each @slide node:

@url built slide Contains the absolute path to the final slide in the _build/html subfolder of the slideshow folder. If present, this @url node completely disables rebuilding the slide.

@url screenshot Contains the absolute path to the original screenshot file. If present, this @url node inhibits taking the screenshot.

@url working file Contains the absolute path to the working file. If present, this @url node disables taking the screenshot, creating the working file. The final output file will be regenerated if the working file is newer than the final output file.

@url final output file Contains the absolute path to the final output file.

Thus, to completely recreate an @slide node, you must delete any of the following nodes that appear as its children:

```
@url screenshot
@url working file
@url built slide
```

Making slides

For each slide, the make-slide and make-slide-show commands do the following:

1. Create a slide.

   If the @slide node contains an @screenshot tree, the plugin appends an .. image:: directive referring to the screenshot to the body text of the @slide node. The plugin also creates a child @image node referring to the screenshot.

2. (Optional) Create a screenshot.

   The plugin creates a screenshot for an @slide node only if the @slide node contains an @screenshot node as a direct child.

   Important: this step has largely been superseded by the @button meld script in LeoDocs.leo.

   Taking a screenshot involves the following steps:

   1. Create the **target outline**: screenshot-setup.leo.

   The target outline contains consists of all the children (and their descendants) of the @screenshot node.

   2. Create the **screenshot**, a bitmap (PNG) file.

      The slide commands take a screen shot of the target outline. The @pause option opens the target outline but does *not* take the screenshot. The user must take the screenshot manually. For more details, see the the options section below.

   3. Convert the screenshot file to a **work file**.

      The work file is an SVG (Scalable Vector Graphics) file: http://www.w3.org/Graphics/SVG/.

   4. (Optional) Edit the work file.

      If the @slide node has a child @edit node, the plugin opens Inkscape so that the user can edit the work file.

   5. Render the **final output file**.

      The plugin calls Inkscape non-interactively to render the final output file (a PNG image) from the work file. If the Python Imaging Library (PIL) is available, this step will use PIL to improve the quality of the final output file.

3. Build the slide using Sphinx.

   After making all files, the plugins runs Sphinx by running `make html' in the slideshow folder. This command creates the final .html files in the _build/html subfolder of the slideshow folder.

4. Create url nodes.

   Depending on options, and already-existing @url nodes, the make-slide and make-slide-show commands may create one or more of the following @url nodes:

```
@url built slide
@url screenshot
@url working file
@url final output file
```

Options and settings

You specify options in the headlines of nodes. **Global options** appear as direct children of @slideshow nodes and apply to all @slide nodes unless overridden by a local option. **Local options** appear as direct children of an @slide node and apply to only to that @slide node.

Global options nodes

The following nodes may appear *either* as a direct child of the @slideshow node or as the direct child of an @slide node.

@sphinx_path = <path> This directory contains the slides directory, and the following files: `conf.py', `Leo4-80-border.jpg', `Makefile' and `make.bat'.

@screenshot_height = <int> The height in pixels of screenshots.

@screenshot_width = <int> The height in pixels of screenshots.

@template_fn = <path> The absolute path to inkscape-template.svg

@title = <any text> The title to use for one slide or the entire slideshow.

@title_pattern = <pattern> The pattern used to generate patterns for one slide or the entire slideshow. The title is computed as follows:

```
d = {
    'slideshow_name':slideshow_name,
    'slide_name':    slide_name,
    'slide_number':  sc.slide_number,
}
title = (pattern % (d)).title()
```

If neither an @title or @title_pattern option node applies, the title is the headline of the @slide node. If this is empty, the default pattern is:

```
'%(slideshow_name)s:%(slide_number)s'
```

@verbose = True/False True (or true or 1): generate informational message. False (or false or 0): suppress informational messages.

@wink_path = <path> This path contains screenshots created by wink. This is used only by the meld-slides command.

Local options nodes

The following nodes are valid only as the direct child of an @slide node.

@callout <any text> Generates a text callout in the working .svg file. An @slide node may have several @callout children.

@edit = True/False If True (or true or 1) the plugin enters Inkscape interactively after taking a screenshot.

@markers = <list of integers> Generates `numbered balls' in the working .svg file.

@pause = True/False If True (or true or 1) the user must take the screenshot manually. Otherwise, the plugin takes the screenshot automatically.

If the slide node contains an @pause node as one of its directive children, the slide commands open the target node, but do *not* take a screen shot.

The user may adjust the screen as desired, for example by selecting menus or showing dialogs. The *user* must then take the screen shot manually. **Important**: the screenshot need not be of Leo--it could be a screenshot of anything on the screen.

As soon as the user closes the target outline, the slide commands look for the screen shot on the clipboard. If found, the slide commands save the screenshot to the screenshot file.

@screenshot The root of a tree that becomes the entire contents of screenshot. No screenshot is taken if this node does not exist.

@select <headline> Causes the given headline in the @screenshot outline to be selected before taking the screenshot.

Settings

@string screenshot-bin = <path to inkscape.exe> The full path to the Inkscape program.

File names

Suppose the @slide node is the n'th @slide node in the @slideshow tree whose sanitized name is `name'. The following files will be created in (relative to) the slideshow directory:

```
slide-n.html.txt:   the slide's rST source.
screenshot-n.png:   the original screenshot.
screenshot-n.svg:   the working file.
slide-n.png:        the final output file.
_build/html/slide-n.html: the final slide.
```

slideshow.py

Support slideshows in Leo outlines.

This plugin defines four new commands:

- next-slide-show: move to the start of the next slide show, or the first slide show if no slide show has been seen yet.
- prev-slide-show: move to the start of the previous slide show, or the first slide show if no slide show has been seen yet.
- next-slide: move to the next slide of a present slide show.
- prev-slide: move to the previous slide of the present slide show.

Slides shows consist of a root @slideshow node with descendant @slide nodes. @slide nodes may be organized via non-@slide nodes that do not appear in the slideshow.

All these commands ignore @ignore trees.

13.3.8 Text formatting

bibtex.py

Manages BibTeX files with Leo.

Create a bibliographic database by putting `@bibtex filename' in a headline. Entries are added as nodes, with `@entrytype key' as the headline, and the contents of the entry in body text. The plugin will automatically insert a template for the entry in the body pane when a new entry is created (hooked to pressing enter when typing the headline text). The templates are defined in dictionary `templates' in the <<globals>> section, by default containing all required fields for every entry.

The file is written by double-clicking the node. Thus the following outline:

```
-@bibtex biblio.bib
 +@book key
  author = {A. Uthor},
  year = 1999
```

will be written in the file `biblio.bib' as:

```
@book{key,
author = {A. Uthor},
year= 1999}
```

Strings are defined in @string nodes and they can contain multiple entries. All @string nodes are written at the start of the file. Thus the following outline:

```
-@bibtext biblio.bib
 +@string
  j1 = {Journal1}
 +@article AUj1
  author = {A. Uthor},
  journal = j1
 +@string
  j2 = {Journal2}
  j3 = {Journal3}
```

Will be written as:

```
@string{j1 = {Journal1}}
@string{j2 = {Journal2}}
@string{j3 = {Journal3}}

@article{AUj1,
author = {A. Uthor},
journal = j1}
```

No error checking is made on the syntax. The entries can be organized under nodes --- if the headline doesn't start with `@', the headline and body text are ignored, but the child nodes are parsed as usual.

BibTeX files can be imported by creating an empty node with `@bibtex filename' in the headline. Double-clicking it will read the file `filename' and parse it into a @bibtex tree. No syntax checking is made, `filename' is expected to be a valid BibTeX file.

dtest.py

Sends code to the doctest module and reports the result.

When the Dtest plugin is enabled, the dtest command is active. Typing:

```
Alt-X dtest
```

will run doctest on a file consisting of the current node and it's children. If text is selected only the selection is tested.

From Wikipedia:

```
'Doctest' is a module included in the Python programming language's
standard library that allows for easy generation of tests based on
output from the standard Python interpreter.
```

http://tinyurl.com/cqh53 - Python.org doctest page

http://tinyurl.com/pxhlq - Jim Fulton's presentation:

```
Literate Testing:
Automated Testing with doctest
```

leo_to_html.py

Converts a leo outline to an html web page.

This plugin takes an outline stored in LEO and converts it to html which is then either saved in a file or shown in a browser. It is based on the original leoToHTML 1.0 plugin by Dan Rahmel which had bullet list code by Mike Crowe.

The outline can be represented as a bullet list, a numbered list or using html <h?> type headings. Optionally, the body text may be included in the output.

If desired, only the current node will be included in the output rather than the entire outline.

An xhtml header may be included in the output, in which case the code will be valid XHTML 1.0 Strict.

The plugin is fully scriptable as all its functionality is available through a Leo_to_HTML object which can be imported and used in scripts.

Menu items and @settings

If this plugin loads properly, the following menu items should appear in your File > Export... menu in Leo:

```
Save Outline as HTML  (equivalent to export-html)
Save Node as HTML     (equivalent to export-html-node)
Show Outline as HTML  (equivalent to show-html)
Show Node as HTML     (equivalent to show-html-node)
```

Unless the following appears in an @setting tree:

```
@bool leo_to_html_no_menus = True
```

in which case the menus will **not** be created. This is so that the user can use @menu and @item to decide which commands will appear in the menu and where.

Commands

Several commands will also be made available

export-html will export to a file according to current settings.

export-html-* will export to a file using bullet type `*' which can be **number**, **bullet** or **head**.

The following commands will start a browser showing the html.

show-html will show the outline according to current settings.

show-html-* will show the outline using bullet type `*' which can be **number**, **bullet** or **head**.

The following commands are the same as above except only the current node is converted:

```
export-html-node
export-html-node-*
show-html-node
show-html-node-*
```

Properties

There are several settings that can appear in the leo_to_html.ini properties file in leo's plugins folder or be set via the Plugins > leo_to_html > Properties... menu. These are:

exportpath: The path to the folder where you want to store the generated html file. Default: c:\

flagjustheadlines: Default: `Yes' to include only headlines in the output.

flagignorefiles: Default: `Yes' to ignore @file nodes.

use_xhtml: Yes to include xhtml doctype declarations and make the file valid XHTML 1.0 Strict. Otherwise only a simple <html> tag is used although the output will be xhtml compliant otherwise. Default: Yes

bullet_type: If this is `bullet' then the output will be in the form of a bulleted list. If this is `number' then the output will be in the form of a numbered list. If this is `heading' then the output will use <h?> style headers.

Anything else will result in <h?> type tags being used where `?' will be a digit starting at 1 and increasing up to a maximum of six depending on depth of nesting. Default: number

browser_command: Set this to the command needed to launch a browser on your system or leave it blank to use your systems default browser.

If this is an empty string or the browser can not be launched using this command then python's *webbrowser* module will be tried. Using a bad command here will slow down the launch of the default browser, better to leave it blank. Default: empty string

Configuration

At present, the file leo/plugins/leo_to_html.ini contains configuration settings. In particular, the default export path, ``c:'' must be changed for *nix systems.

leo_to_rtf.py

Outputs a Leo outline as a numbered list to an RTF file. The RTF file can be loaded into Microsoft Word and formatted as a proper outline.

If this plug-in loads properly, you should have an ``Outline to Microsoft RTF'' option added to your File > Export... menu in Leo.

Settings such as outputting just the headlines (vs. headlines & body text) and whether to include or ignore the contents of @file nodes are stored in the rtf_export.ini file in your Leoplugins folder.

The default export path is also stored in the INI file. By default, it's set to c:so you may need to modify it depending on your system.

13.3.9 User interface

UNL.py

Supports Uniform Node Locators (UNL's) for linking to nodes in any Leo file.

UNL's specify nodes within any Leo file. You can use them to create cross-Leo-file links! UNL

This plugin consists of two parts:

1. Selecting a node shows the UNL in the status line at the bottom of the Leo window. You can copy from the status line and paste it into headlines, emails, whatever.

2. Double-clicking @url nodes containing UNL's select the node specified in the UNL. If the UNL species in another Leo file, the other file will be opened.

Format of UNL's:

UNL's referring to nodes within the present outline have the form:

```
headline1-->headline2-->...-->headlineN
```

headline1 is the headline of a top-level node, and each successive headline is the headline of a child node.

UNL's of the form:

```
file:<path>#headline1-->...-->headlineN
```

refer to a node specified in <path> For example, double clicking the following headline will take you to Chapter 8 of Leo's Users Guide:

```
@url file:c:/prog/leoCvs/leo/doc/leoDocs.leo#Users Guide-->Chapter 8: Customizing Leo
```

For example, suppose you want to email someone with comments about a Leo file. Create a comments.leo file containing @url UNL nodes. That is, headlines are @url followed by a UNL. The body text contains your comments about the nodes in the _other_ Leo file! Send the comments.leo to your friend, who can use the comments.leo file to quickly navigate to the various nodes you are talking about. As another example, you can copy UNL's into emails. The recipient can navigate to the nodes `by hand' by following the arrows in the UNL.

Notes:

- At present, UNL's refer to nodes by their position in the outline. Moving a node will break the link.
- Don't refer to nodes that contain UNL's in the headline. Instead, refer to the parent or child of such nodes.
- You don't have to replace spaces in URL's or UNL's by `%20'.

chapter_hoist.py

Creates hoist buttons.

This plugin puts two buttons in the icon area: a button called `Save Hoist' and a button called `Dehoist'. The `Save Hoist' button hoists the presently selected node and creates a button which can later rehoist the same node. The `Dehoist' button performs one level of dehoisting

Requires at least version 0.19 of mod_scripting.

detect_urls.py

Colorizes URLs everywhere in node's body on node selection or saving. Double click on any URL launches it in default browser.

URL regex: (http|https|file|ftp)://[^s"']+[w=/]

Related plugins: color_markup.py; rClick.py

EditAttributes.py

Lets the user to associate text with a specific node.

Summon it by pressing button-2 or button-3 on an icon Box in the outline. This will create an attribute editor where the user can add, remove and edit attributes. Since attributes use the underlying tnode, clones will share the attributes of one another.

interact.py

Adds buttons so Leo can interact with command line environments.

> **20100226** see also leoscreen.py for a simpler approach.

Currently implements *bash* shell and *psql* (postresql SQL db shell).

Single-line commands can be entered in the headline with a blank body, multi-line commands can be entered in the body with a descriptive title in the headline. Press the *bash* or *psql* button to send the command to the appropriate interpreter.

The output from the command is **always** stored in a new node added as the first child of the command node. For multi-line commands this new node is selected. For single-line command this new node is not shown, instead the body text of the command node is updated to reflect the most recent output. Comment delimiter magic is used to allow single-line and multi-line commands to maintain their single-line and multi-line flavors.

Both the new child nodes and the updated body text of single-line commands are timestamped.

For the *bash* button the execution directory is either the directory containing the *.leo* file, or any other path as specified by ancestor *@path* nodes.

Currently the *psql* button just connects to the default database. ";" is required at the end of SQL statements.

Requires *pexpect* module.

maximizeNewWindows.py

Maximizes all new windows.

mod_framesize.py

Sets a hard coded frame size.

Prevents Leo from setting custom frame size (e.g. from an external .leo document)

plugins_menu.py

Creates a Plugins menu and adds all actives plugins to it.

Selecting these menu items will bring up a short **About Plugin** dialog with the details of the plugin. In some circumstances a submenu will be created instead and an `About' menu entry will be created in this.

INI files and the Properties Dialog

If a file exists in the plugins directory with the same file name as the plugin but with a .ini extension instead of .py, then a **Properties** item will be created in a submenu. Selecting this item will pop up a Properties Dialog which will allow the contents of this file to be edited.

The .ini file should be formated for use by the python ConfigParser class.

Special Methods

Certain methods defined at the top level are considered special.

cmd_XZY If a method is defined at the module level with a name of the form **cmd_XZY** then a menu item **XZY** will be created which will invoke **cmd_XZY** when it is selected. These menus will appear in a sub menu.

applyConfiguration

topLevelMenu This method, if it exists, will be called when the user clicks on the plugin name in the plugins menu (or the **About** item in its submenu), but only if the plugin was loaded properly and registered with g.plugin_signon.

Special Variable Names

Some names defined at the top level have special significance.

__plugin_name__ This will be used to define the name of the plugin and will be used as a label for its menu entry.

__plugin_priority__ Plugins can also attempt to select the order they will appear in the menu by defining a __plugin_prioriy__. The menu will be created with the highest priority items first. This behavior is not guaranteed since other plugins can define any priority. This priority does not affect the order of calling handlers. To change the order select a number outside the range 0-200 since this range is used internally for sorting alphabetically. Properties and INI files.

redirect_to_log.py

Sends all output to the log pane.

scripts_menu.py

Creates a Scripts menu for LeoPy.leo.

zenity_file_dialogs.py

Replaces the gui file dialogs on Linux with external calls to the zenity gtk dialog package.

This plugin is more a proof of concept demo than a useful tool. The dialogs presented do not take filters and starting folders can not be specified.

Despite this, some Linux users might prefer it to the gui dialogs.

13.4 Qt only plugins

13.4.1 attrib_edit.py

Edits user attributes in a Qt frame.

This plugin creates a frame for editing attributes similar to:

```
Name:   Fred Blogs
Home:   555-555-5555
Work:   555-555-5556
```

attrib_edit is also intended to provide attribute editing for other plugins, see below.

The attributes can be stored in different ways, three modes are implemented currently:

v.u mode These attributes are stored in the ``unknownAttributes" (uA) data for each node, accessed via v.u.

Field: Attributes are lines starting (no whitespace) with ``AttributeName:" in the body text.

@Child Attributes are the head strings of child nodes when the head string starts with `@AttributeName` where the first letter (second character) must be capitalized.

The plugin defines the following commands, available either in the plugin's sub-menu in the Plugins menu, or as Alt-X attrib-edit-*.

attrib-edit-modes Select which attribute setting / getting modes to use. More than one mode can be used at the same time.

You can also control which modes are active by listing them with the @data attrib_edit_active_modes setting. For example:

```
Field:
@Child
# v.u mode
```

would cause only the ``Field:" and ``@Child" modes to be active be default.

attrib-edit-manage Select which attributes, from all attributes seen so far in this outline, to include on the current node.

attrib-edit-scan Scan the entire outline for attributes so attrib-edit-manage has the complete list.

attrib-edit-create Create a new attribute on the current node. If Field: or @Child modes are active, they simply remind you how to create an attribute in the log pane. If the ``v.u mode" mode is active, you're prompted for a path for the attribute. For example:

```
addressbook First
```

to store the attribute in v.u['addressbook']['_edit']['First']

As a convenience, entering a path like:

```
todo metadata created|creator|revised
```

would create:

```
        v.u.['todo']['metadata']['_edit']['created']
        v.u.['todo']['metadata']['_edit']['creator']
        v.u.['todo']['metadata']['_edit']['revised']
```

Technical details

See the source for complete documentation for use with other plugins. Here are some points of interest:

- In addition to v.u['addressbook']['_edit']['first'], paths like v.u['addressbook']['_edit']['_int']['age'] may be used to identify type, although currently there's no difference in the edit widget.
- In the future the plugin may allow other plugins to register to provide attribute path information, instead of just scanning for ['_edit'] entries in v.u.
- Currently there's no sorting of the attributes in ``v.u mode", which is a problem for some applications. It's unclear where the desired order would be stored, without even more repetition in v.u. When other plugins can register to manipulate the attribute list each plugin could address this, with unordered presentation in the absence of the client plugin.
- There's code to have the editor appear in a tab instead of its own area under the body editor, but (a) this is always being buried by output in the log window, and (b) there's a bug which leaves some (harmless) ghost widgets in the background. Enable by @setting attrib_edit_placement to `tab'.

13.4.2 colorize_headlines.py

Manipulates appearance of individual tree widget items.

This plugin is mostly an example of how to change the appearance of headlines. As such, it does a relatively mundane chore of highlighting @thin, @auto, @shadow nodes in bold.

13.4.3 contextmenu.py

Defines various useful actions for context menus (Qt only).

Examples are:

- Edit in $EDITOR
- Edit @thin node in $EDITOR (remember to do ``refresh" after this!)
- Refresh @thin node from disk (e.g. after editing it in external editor)
- Go to clone

Here's an example on how to implement your own context menu items in your plugins:

```python
def nextclone_rclick(c,p, menu):
    """ Go to next clone """

    # only show the item if you are on a clone
    # this is what makes this "context sensitive"
    if not p.isCloned():
        return
```

```
def nextclone_rclick_cb():
    c.goToNextClone()

# 'menu' is a QMenu instance that was created by Leo
# in response to right click on tree item

action = menu.addAction("Go to clone")
action.connect(action, QtCore.SIGNAL("triggered()"), nextclone_rclick_cb)
```

And call this in your plugin *once*:

```
g.tree_popup_handlers.append(nextclone_rclick)
```

13.4.4 nav_qt.py

Adds ``Back" and ``Forward" buttons (Qt only).

Creates ``back" and ``forward" buttons on button bar. These navigate the node history.

This plugin does not need specific setup. If the plugin is loaded, the buttons will be available. The buttons use the icon specified in the active Qt style

13.4.5 projectwizard.py

Creates a wizard that creates @auto nodes.

Opens a file dialog and recursively creates @auto & @path nodes from the path where the selected file is (the selected file itself doesn't matter.)

13.4.6 quicksearch.py

Adds a fast-to-use search widget, like the ``Find in files" feature of many editors.

Just load the plugin, activate ``Nav" tab, enter search text and press enter.

The pattern to search for is, by default, a case *insensitive* fnmatch pattern (e.g. foo*bar), because they are typically easier to type than regexps. If you want to search for a regexp, use `r:' prefix, e.g. r:foo.*bar.

Regexp matching is case sensitive; if you want to do a case-insensitive regular expression search (or any kind of case-sensitive search in the first place), do it by searching for ``r:(?i)Foo". (?i) is a standard feature of Python regular expression syntax, as documented in

http://docs.python.org/library/re.html#regular-expression-syntax

13.4.7 scrolledmessage.py

Provides a Scrolled Message Dialog service for Qt.

The plugin can display messages supplied as plain text or formatted as html. In addition the plugin can accept messages in rst format and convert them to be displayed as html.

The displayed format can be controlled by the user via check boxes, so rst messages may be viewed either as text or as html. Html messages can also be viewed as raw text, which will be a good debug feature when creating complex dynamically generated html messages.

The user interface is provided by a ScrolledMessage.ui file which is dynamically loaded each time a new dialog is loaded.

The dialog is not modal and many dialogs can exist at one time. Dialogs can be named and output directed to a dialog with a specific name.

The plugin is invoked like this:

```
g.doHook('scrolledMessage', c=c, msg='message', title='title',  ...etc    )
```

or:

```
g.app.gui.runScrolledMessageDialog(c=c, ...etc)
```

All parameters are optional except c.

Parameters

msg: The text to be displayed (html, rst, plain).

If the text starts with `rst:' it is assumed to be rst text and is converted to html for display after the rst: prefix has been removed. If the text starts with `<' it is assumed to be html. These auto detection features can be overridden by `flags'.

label: The text to appear in a label above the display. If it is `', the label is hidden.

title: The title to appear on the window or dock.

flags: Says what kind of message: `rst', `text', `html'. This overrides auto-detection.

Flags can be combined, for example, `rst html' causes the message to be interpreted as rst and displayed as html.

13.4.8 spydershell.py

Launches the spyder environment with access to Leo instance. See http://packages.python.org/spyder/

Execute alt-x spyder-launch to start spyder. Execute alt-x spyder-update to pass current c,p,g to spyder interactive session. spyder-update also shows the window if it was closed before.

13.4.9 stickynotes.py

Adds simple ``sticky notes'' feature (popout editors) for Qt gui.

Adds the following (Alt-X) commands:

stickynote pop out current node as a sticky note

stickynoter pop out current node as a rich text note

stickynoteenc pop out current node as an encrypted note

stickynoteenckey enter a new en/decryption key

tabula add the current node to the stickynotes in the *Tabula* sticky note dock window, and show the window

tabula-show show the`Tabula` sticky note dock window (without adding the current node)

tabula-marked add all marked nodes to the stickynotes in the *Tabula* sticky note dock window, and show the window

Sticky notes are synchronized (both ways) with their parent Leo node.

Encrypted mode requires the python-crypto module.

The first time you open a note in encrypted mode you'll be asked for a pass phrase. That phrase will be used for the rest of the session, you can change it with Alt-X stickynoteenckey, but probably won't need to.

The encrypted note is stored in base64 encoded *encrypted* text in the parent Leo node, if you forget the pass phrase there's no way to un-encrypt it again. Also, you must not edit the text in the Leo node.

When **creating an encrypted note**, you should **start with an empty node**. If you want to encrypt text that already exists in a node, select-all cut it to empty the node, then paste it into the note.

13.4.10 todo.py

Provides to-do list and simple task management for leo (Qt only).

This plugin adds time required, progress and priority settings for nodes. With the @project tag a branch can display progress and time required with dynamic hierarchical updates.

For full documentation see:

- http://leo.zwiki.org/ToDo
- http://leo.zwiki.org/tododoc.html

13.4.11 viewrendered.py

Creates a window for *live* rendering of rst, html, etc. Qt only.

viewrendered.py creates a single Alt-X style command, viewrendered, which opens a new window where the current body text is rendered as HTML (if it starts with `<'), or otherwise reStructuredText. reStructuredText errors and warnings may be shown. For example, both:

```
Heading
-------

`This` is **really** a line of text.
```

and:

```
<h1>Heading<h1>

<tt>This</tt> is <b>really</b> a line of text.
```

will look something like:

Heading

This is **really** a line of text.

13.4.12 graphcanvas.py

Adds a graph layout for nodes in a tab. Requires Qt and the backlink.py plugin.

第 14 章

Writing Plugins

A **plugin** is a Python file that appears in Leo's plugin directory. Plugins modify how Leo works. With plugins you can give Leo new commands, modify how existing commands work, or change any other aspect of Leo's look and feel. leoPlugins.leo contains all of Leo's official plugins. Studying this file is a good way to learn how to write plugins.

You **enable** plugins using @enabled-plugins nodes in leoSettings.leo or myLeoSettings.leo. For more details, see the @enabled-plugins node in leoSettings.leo. Leo imports all enabled plugins at startup time. Plugins become **active** if importing the plugin was successful.

Writing plugins is quite similar to writing any other Leo script. See Scripting Leo with Python. In particular:

1. Plugins can use any of Leo's source code simply by importing any module defined in leoPy.leo.
2. Plugins can register event handlers just like any other Leo script. For full details, see the section called event handlers in Leo's scripting chapter.

The rest of this chapters discusses topics related specifically to plugins.

Contents

- Writing Plugins
  - enabled-plugins
  - Support for unit testing
  - Important security warnings

14.1 enabled-plugins

@enabled-plugins node is as list of plugins to load. If you have @enabled-plugins node in your myLeoSettings.leo, the plugins are loaded from there. If such a node doesn't exist, the global leoSettings.leo is used instead.

The @enabled-plugins bundled in leoSettings.leo contains a list of default (recommended) plugins. For your own @enabled-plugins in myLeoSettings.leo, you should use the node in leoSettings.leo as a starting point unless you are certain you want to disable a recommended plugin.

@enabled-plugins nodes contain the list of enabled plugins, one per line.

Comment lines starting with `#' are ignored.

Plugins are essentially normal python modules, and loading a plugin basically means importing it and running the ``init'' function in the module's root level namespace. A line in @enabled-plugins is a module name that leo should import.

Here's an example @enabled-plugins node:

```
# Standard plugins enabled in official distributions....

plugins_menu.py
quicksearch.py

# third party plugins

# 'leoplugin' module inside python package 'foo'
foo.leoplugin

# top-level module
barplugin
```

Note that some entries end with .py. This is done to retain backwards compatibility - if an entry ends with .py, it means a plugin in Leo's `plugins' directory (package) and is translated to e.g. ``leo.plugins.plugins_menu'' before importing.

Normally, a third party plugin should be a basic python module that is installed globally for the python interpreter with ``python setup.py install''. Installing plugins to Leo's `plugins' directory is not recommended, as such plugins can disappear when Leo is upgraded.

14.2 Support for unit testing

The plugins test suite creates a new convention: if a plugin has a function at the outer (module) level called unitTest, Leo will call that function when doing unit testing for plugins. So it would be good if writers of plugins would create such a unitTest function. To indicate a failure the unitTest just throws an exception. Leo's plugins test suite takes care of the rest.

14.3 Important security warnings

Naively using plugins can expose you and your .leo files to malicious attacks. The fundamental principles are:

```
Scripts and plugins must never blindly execute code from untrusted sources.
```

and:

```
.leo files obtained from other people may potentially contain hostile code.
```

Stephen Schaefer summarizes the danger this way:

```
I foresee a future in which the majority of leo projects come from
marginally trusted sources...a world of leo documents sent hither and yon -
resumes, project proposals, textbooks, magazines, contracts - and as a race
of Pandora's, we cannot resist wanting to see "What's in the box?" And are
we going to fire up a text editor to make a detailed examination of the
ASCII XML? Never! We're going to double click on the cute leo file icon, and
leo will fire up in all its raging glory. Just like Word (and its macros) or
Excel (and its macros).
```

In other words:

```
When we share "our" .leo files we can NOT assume that
we know what is in our "own" documents!
```

Not all environments are untrustworthy. Code in a commercial cvs repository is probably trustworthy: employees might be terminated for posting malicious code. Still, the potential for abuse exists anywhere.

In Python it is very easy to write a script that will blindly execute other scripts:

```python
# Warning: extremely dangerous code

# Execute the body text of all nodes that start with `@script`.
def onLoadFile():
    for p in c.all_positions():
        h = p.h.lower()
        if g.match_word(h,0,"@script"):
            s = p.b
            if s and len(s) > 0:
                try: # SECURITY BREACH: s may be malicious!
                    exec(s + '\n')
                except:
                    es_exception()
```

Executing this kind of code is typically an intolerable security risk. **Important**: rexec provides *no protection whatever*. Leo is a repository of source code, so any text operation is potentially malicious. For example, consider the following script, which is valid in rexec mode:

```
badNode = c.p
for p in c.all_positions():
    << change `rexec` to `exec` in p's body >>
<< delete badNode >>
<< clear the undo stack >>
```

This script will introduce a security hole the .leo file without doing anything prohibited by rexec, and without leaving any traces of the perpetrating script behind. The damage will become permanent *outside* this script when the user saves the .leo file.

第 **15** 章

Unit testing with Leo

This chapter describes how you can execute Python unit test from within Leo outlines.

Leo's **unit test commands** run the unit tests created by @test and @suite nodes. run-unit-tests and run-unit-tests-locally run all unit tests in the presently selected part of the Leo outline; run-all-unit-tests and run-all-unit-tests-locally run all unit tests in the entire Leo outline.

Important: you must run Leo in a console window to see the output the unit tests. Leo's unit test commands run all the unit tests using the standard unittest text test runner, and the output of the unit tests appears in the console.

test/unitTest.leo contains many examples of using @test and @suite nodes.

Contents

- Unit testing with Leo
  - Using @test nodes
  - Using @suite nodes
  - Using @mark-for-unit-tests
  - How the unit test commands work
  - @button timer
  - @button profile

15.1 Using @test nodes

@test nodes are nodes whose headlines start with @test. The unit test commands convert the body text of @test nodes into a unit test automatically. That is, Leo's unit test commands automatically create a unittest.TestCase instances which run the body text of the @test node. For example, let us consider one of Leo's actual unit tests. The headline is:

```
@test consistency of back/next links
```

The body text is:

```
if g.unitTesting:
    c,p = g.getTestVars() # Optional: prevents pychecker warnings.
    for p in c.all_positions():
        back = p.back()
        next = p.next()
        if back: assert(back.getNext() == p)
        if next: assert(next.getBack() == p)
```

When either of Leo's unit test commands finds this @test node the command will run a unit test equivalent to the following:

```
import leo.core.leoGlobals as g

class aTestCase (unittest.TestCase):
    def shortDescription():
        return '@test consistency of back/next links'
    def runTest():
        c,p = g.getTestVars()
        for p in c.all_positions():
            back = p.back()
            next = p.next()
            if back: assert(back.getNext() == p)
            if next: assert(next.getBack() == p)
```

As you can see, using @test nodes saves a lot of typing:

- You don't have to define a subclass of unittest.TestCase.
- Within your unit test, the c, g and p variables are predefined, just like in Leo scripts.
- The entire headline of the @test node becomes the short description of the unit test.

Important note: notice that the first line of the body text is a **guard line**:

```
if g.unitTesting:
```

This guard line is needed because this particular @test node is contained in the file leoNodes.py. @test nodes that appear outside of Python source files do not need guard lines. The guard line prevents the unit testing code from being executed when Python imports the leoNodes module; the g.unitTesting variable is True only while running unit tests.

New in Leo 4.6: When Leo runs unit tests, Leo predefines the `self' variable to be the instance of the test itself, that is an instance of unittest.TestCase. This allows you to use methods such as self.assertTrue in @test and @suite nodes.

Note: Leo predefines the c, g, and p variables in @test and @suite nodes, just like in other scripts. Thus, the line:

```
c,p = g.getTestVars()
```

is not needed. However, it prevents pychecker warnings that c and p are undefined.

15.2 Using @suite nodes

@suite nodes are nodes whose headlines start with @suite. @suite nodes allow you to create and run custom subclasses of unittest.TestCase.

Leo's test commands assume that the body of an suite node is a script that creates a suite of tests and places that suite in g.app.scriptDict['suite']. Something like this:

```
if g.unitTesting:
    __pychecker__ = '--no-reimport' # Prevents pychecker complaint.
    import unittest
    c,p = g.getTestVars() # Optional.
    suite = unittest.makeSuite(unittest.TestCase)
    << add one or more tests (instances of unittest.TestCase) to suite >>
    g.app.scriptDict['suite'] = suite
```

Note: as in @test nodes, the guard line, `if unitTesting:', is needed only if the @suite node appears in a Python source file.

Leo's test commands first execute the script and then run suite in g.app.scriptDict.get(`suite') using the standard unittest text runner.

You can organize the script in an @suite nodes just as usual using @others, section references, etc. For example:

```
if g.unitTesting:
    __pychecker__ = '--no-reimport'
    import unittest
    c,p = g.getTestVars() # Optional.
    # children define test1,test2..., subclasses of unittest.TestCase.
    @others
    suite = unittest.makeSuite(unittest.TestCase)
    for test in (test1,test2,test3,test4):
        suite.addTest(test)
    g.app.scriptDict['suite'] = suite
```

15.3 Using @mark-for-unit-tests

When running unit tests externally, Leo copies any @mark-for-unit-tests nodes to dynamicUnitTest.leo. Of course, this is in addition to all @test nodes and @suite nodes that are to be executed. You can use @mark-for-unit-test nodes to include any ``supporting data" you want, including, say, ``@common test code" to be imported as follows:

```
exec(g.findTestScript(c,'@common test code'))
```

Note: putting @settings trees as descendants of an @mark-for-unit-test node will copy the @setting tree, but will *not* actually set the corresponding settings.

15.4 How the unit test commands work

The run-all-unit-tests-locally and run-unit-tests-locally commands run unit tests in the process that is running Leo. These commands *can* change the outline containing the unit tests.

The run-all-unit-tests and run-unit-tests commands run all tests in a separate process, so unit tests can never have any side effects. These commands never changes the outline from which the tests were run. These commands do the following:

1. Copy all @test, @suite, @unit-tests and @mark-for-unit-test nodes (including their descendants) to the file test/dynamicUnitTest.leo.
2. Run test/leoDynamicTest.py in a separate process.
   - leoDynamicTest.py opens dynamicUnitTest.leo with the leoBridge module. Thus, all unit tests get run with the nullGui in effect.
   - After opening dynamicUnitTest.leo, leoDynamicTest.py runs all unit tests by executing the leoTest.doTests function.
   - The leoTests.doTests function searches for @test and @suite nodes and processes them generally as described above. The details are a bit different from as described, but they usually don't matter. If you *really* care, see the source code for leoTests.doTests.

15.5 @button timer

The timit button in unitTest.leo allows you to apply Python's timeit module. See http://docs.python.org/lib/module-timeit.html. The contents of @button timer is:

```
import leo.core.leoTest as leoTest
leoTest.runTimerOnNode(c,p,count=100)
```

runTimerOnNode executes the script in the presently selected node using timit.Timer and prints the results.

15.6 @button profile

The profile button in unitTest.leo allows you to profile nodes using Python's profiler module. See http://docs.python.org/lib/module-profile.html The contents of @button profile is:

```
import leo.core.leoTest as leoTest
leoTest.runProfileOnNode(p,outputPath=None) # Defaults to leo\test\profileStats.txt
```

runProfileOnNode runs the Python profiler on the script in the selected node, then reports the stats.

第 16 章

Debugging with Leo

This chapter discusses debugging Python scripts with Leo. Be aware of the distinction between **Leo-specific** scripts and **general** scripts. Leo-specific scripts access data in the Leo outline in which they are contained; general scripts do not.

Contents

- Debugging with Leo
  - Using g.trace and g.pdb
  - Settings for winpdb
  - Debugging scripts with winpdb
    * The debug command
    * The execute-script command with explicit debugger breaks

16.1 Using g.trace and g.pdb

Inserting g.trace statements in my Python code is usually my first debugging choice. The g.trace statement prints the name of the function in which the call to g.trace occurs, followed by the value of its arguments. The output of the g.trace goes to the console, so you must run Leo in a console window to use g.trace.

Inserting and deleting g.trace statements is fast, provided that your work flow makes it easy to restart the program under test. As a result, using g.trace statements is similar to setting tracepoints in a debugger, with the advantage that (disabled) tracepoints remain in the source code for future use. You will find many examples of using g.trace throughout Leo's source code.

My second choice is using g.pdb to set breakpoints for the pdb debugger. Pdb uses the console for all interaction, so you must run Leo in a console window. See the FAQ for a discussion of both g.trace and g.pdb.

16.2 Settings for winpdb

The following settings in leoSettings.leo control debugger operation. The settings shown here will be assumed to be in effect throughout this chapter:

```
@string debugger_kind = winpdb
```

This setting controls what debugger the `Debug Script' script button uses. Eventually this setting will control what debugger the debug command uses:: At present the only valid value is `winpdb'

> @bool write_script_file = True

True: The execute script command writes the script to be executed to a file, then executes the script using Python's execFile function. The script_file_path setting specifies the path to this file. False (legacy): The execute script command uses Python's exec command to execute the script.

@string script_file_path = ../test/scriptFile.py

The path to the file to be written by the execute-script command. Notes:

- This setting has effect only if the write_script_file setting is True.
- Use / as the path delimiter, regardless of platform.
- The default path is ../test/scriptFile.py if no path is given.
- The path starts at g.app.loadDir, so for example ../test/scriptFile.py is equivalent to leo/test/scriptFile.py.
- The filename should end in .py.

@string debugger_path = None

16.3 Debugging scripts with winpdb

The following three section discuss three ways of debugging scripts with winpdb. The first two sections tell how to debug general scripts; the last section tells how to debug Leo-specific scripts.

winpdb and its documentation have been improved recently. For more details, see the embedded winpdb docs. The discussion of embedded debugging may have been written specifically with Leo in mind.

16.3.1 The debug command

This way of debugging can only be used for general scripts, not leo-specific scripts. The debug command writes the script to scriptFile.py and invokes winpdb. winpdb opens and is already `attached' to the script to be debugged. You can single-step as you like. Leo continues to run, but killing the debugger will also kill Leo.

16.3.2 The execute-script command with explicit debugger breaks

This way of debugging scripts allows winpdb to debug scripts that use c, g and p. A bit more work is needed because winpdb does not start automatically. Here are step-by step instructions:

1. Insert the following two lines of code at the start of the script to be debugged:

```
import rpdb2
rpdb2.start_embedded_debugger('go',fAllowUnencrypted=True)
```

2. Execute Leo's execute-script command (*not* the debug command). Leo will appear to hang: start_embedded_debugger is waiting for *another* copy of winpdb to `attach' to the script's process. The default timeout is 5 minutes, after which an exception gets thrown.

3. Start winpdb explicitly by executing something like the following in a console:

```
python /Python26/Scripts/_winpdb.py -t
```

   The -t option tells winpdb that no encoding of password is necessary. The password is specified in the call to rpdb2.start_embedded_debugger in your script. In our example, the password is `go'.

4. Use winpdb's File:Attach command to attach winpdb to Leo. Specify the password as `go' and you will see the scriptFile.py containing your entire script. You can now execute or single-step through the script. To repeat, c, g and p are defined, so you can debug any script this way.

第 **17** 章

Using @shadow

This chapter describes an important new feature that debuted in Leo 4.5 b2: @shadow trees. These trees combine the benefits of @auto, @file and @nosent trees:

- The (public) files created by @shadow trees contain no sentinels, but
- Leo is able to update @shadow trees in the Leo outline based on changes made to public files outside of Leo.

@shadow trees are often useful for studying or editing source files from projects that don't use Leo. In such situations, it is convenient to import the @shadow tree from the (public) sources. As discussed below, Leo can import @shadow trees automatically, using the same algorithms used by @auto trees.

The crucial ideas and algorithms underlying @shadow trees are the invention of Bernhard Mulder.

Contents

- Using @shadow
  - Overview
  - Creating @shadow trees
  - What the update algorithm does
  - Aha: boundary cases don't matter

17.1 Overview

Using @shadow trees is the best choice when you want to have the full power of Leo's outlines, but wish to retain the source files in their original format, without Leo sentinels (markup) in comments in the source file.

Leo's @file trees create external files containing comments called sentinels. These sentinel lines allow Leo to recreate the outlines structure of @file trees. Alas, many people and organizations find these added sentinel lines unacceptable. @nosent nodes create external files without sentinels, but at a cost: Leo can not update @nosent trees when the corresponding external file is changed outside of Leo.

@shadow trees provide a way around this dilemma. When Leo saves an @shadow tree, it saves two copies of the tree: a **public** file without sentinels, and a **private** file containing sentinels. Using Bernhard Mulder's brilliant **update algorithm**, Leo is able to update @shadow trees in the Leo outline based *solely* on changes to public files.

Leo writes private files to a subfolder of the folder containing the public file: by default this folder is called .leo_shadow. You can change the name of this folder using the @string shadow_subdir setting. Note that private files need not be known to source code control systems such as bzr or cvs.

That's *almost* all there is to it. The following sections discuss important details:

- How to create @shadow trees.
- How @shadow works.
- Why the update algorithm is sound.

17.2 Creating @shadow trees

The first step in creating an @shadow tree is to create a node whose headline is @shadow <*filename*>.

Thus, you can create an @shadow node and save your outline, regardless of whether the original file exists. The next time Leo reads the @shadow node, Leo will **create** the entire @shadow tree using the same logic as for @auto trees. You can cause Leo to read the @shadow node in two ways: 1) by closing and reloading the Leo outline or 2) by selecting the @shadow node and executing the File:Read/Write:Read @shadow Node command.

Important: Leo imports the private file into the @shadow tree only if

1. the public file exists and
2. the private file does *not* exist.

Thus, Leo will import code into each @shadow node at most once. After the first import, updates are made using the update algorithm.

Note: just as for @auto, Leo will never read (import) or write an @shadow tree if the @shadow node is under the influence of an @ignore directive.

17.3 What the update algorithm does

Suppose our @shadow tree is @shadow a.py. When Leo writes this tree it creates a public file, a.py, and a private file, .leo_shadow/xa.p (or just xa.p for short). Public files might can committed to a source code control system such as cvs or bzr. Private files should *not* be known to cvs or bzr.

Now suppose a.py has been changed outside of Leo, say as the result of a bzr merge. The corresponding private file, xa.p, will *not* have been changed. (Private files should *never* change outside of Leo.

When Leo reads the *new* (and possibly updated) public file it does the following:

1. Recreates the *old* public file by removing sentinels from the (unchanged!) *private* file.

2. Creates a set of diffs between the old and new *public* files.
3. Uses the diffs to create a new version of the *private* file.
4. Creates the @shadow tree using the new *private* file.

Important: The update algorithm never changes sentinels. This means that the update algorithm never inserts or deletes nodes. The user is responsible for creating nodes to hold new lines, or for deleting nodes that become empty as the result of deleting lines.

Step 3 is the clever part. To see all the details of how the algorithm works, please study the x.propagate_changed_lines method in leoShadow.py. This code is heavily commented.

17.4 Aha: boundary cases don't matter

There are several boundary cases that the update algorithm can not resolve. For example, if a line is inserted at the boundary between nodes, the updated algorithm can not determine whether the line should be inserted at the end of one node of the start of the next node.

Happily, the inability of the update algorithm to distinguish between these two cases **does not matter**, for three very important reasons:

1. No matter which choice is made, the *public* file that results is the same. **The choice doesn't matter**, so the update algorithm is completely and absolutely safe.
2. Leo reports any nodes that were changed as the result of the update algorithm. In essence, these reports are exactly the same as the reports Leo makes when @file trees were changed as the result of changes made externally to Leo. It is as easy for the user to review changes to @shadow trees as it is to review changes to @thin or @file trees.
3. Suppose the user moves a line from the end of one node to the beginning of the following node, or vice versa. Once the user saves the file, the *private* file records the location of the moved line. The next time the user reads the @shadow file, the line will *not* be subject to the update algorithm because the line has not been changed externally. The location of the line (on the boundary) will be completely determined and it will never need to be moved across the boundary.

Understanding these three reasons finally convinced me that @shadow could be made to work reliably.

第 18 章

The leoInspect Module

The leoInspect module provides answers to questions about Python source code such as:

- Where are all assignments to `w' in leoEditCommands.py?
- Which of those assignments are ``unusual'' in some way?

The leoInspect module grew out of a re-imagining of the new-pylint project, which has been a ``hobby'' project of mine for several years. Rather than attempting global ``proofs'' of difficult propositions, as new-pylint does, the leoInspect module answers specific questions about modules, functions, classes, methods and statements. We can use such answers while debugging, or as documentation, or especially as the foundation for *fast* unit tests.

The leoInspect module provides answers to questions about Python source code. leoInspect is an elegant and easy-to-use front end for Python's AST (Abstract Syntax Tree) trees *and* a window into a richly connected set of semantic data built *from* AST trees.

The simplicity of the leoInspect module could be called ``third generation'' simplicity. Several implementation Ahas lie behind it.

Contents

- The leoInspect Module
  - A query language
  - Comparing leoInspect and Pylint
  - Still to do
  - Theory of operation

18.1 A query language

Mathematica's expressions inspired the design of the query language: simple and task oriented. All details in the background. leoInspect makes *Python* the query language! Let's see how.

All queries start with a call to leoInspect.module:

```
import leo.core.leoInspect as inspect
o = inspect.module('leoApp.py')
```

The call to inspect.module creates a **query object** o representing all the data contained in the AST for leoApp.py. But this query object *also* represents a **context**, a module, class, function, method or statement.

For any context o, we can use **getters** to get lists of other contexts:

```
aList = o.calls()      # All function/method calls in context o.
aList = o.classes()    # All classes in context o.
aList = o.functions()  # All functions in context o.
aList = o.statements() # All the statements in context o.
```

Assignments and calls are especially important for queries. The following getters return the assignments related to some name s:

```
aList = o.assignments_to(s)
aList = o.assignments_using(s)
aList = o.calls_to(s)
aList = o.call_args_of(s)
```

The o.name getter returns the name of any module, class, method, function or var:

```
s = o.name()
```

The o.format getter returns a human-readable representation of a context's AST tree:

```
s = o.format()
```

These getters generally make it unnecessary to access AST trees directly: ASTs are merely part of the ``plumbing'' of the leoInspect module. However, the o.tree getter returns the actual AST tree if you really need it:

```
ast_tree = o.tree()
```

Let's see how to create actual queries. Here is a script to discover all assignments to `w' in leoEditCommands.py is:

```
import leo.core.leoInspect as inspect
m = inspect.module('leoEditCommands.py')
for a in m.assignments_to('w'):
    print(a.format())
```

It is easy to ``zero in'' on particular classes or method using the o.name getter. The following script prints all assignments to the `files' ivar of the LoadManager class in leoApp.py:

```
import leo.core.leoInspect as inspect
m = inspect.module('leoApp.py')
```

```
for theClass in m.classes():
    if theClass.name() == 'LoadManager':
        for a in m.assignments_to('files'):
            print(a.format())
```

We have seen that the getters hide all the messy details of Python AST trees. This is a revolution in using AST trees!

18.2 Comparing leoInspect and Pylint

Could one create a lint-like programming using leoInspect? Perhaps, but much more work would be required. Indeed, pylint is an extremely capable program. It can make complex deductions that are presently far beyond leoInspect's capabilities.

However, leoInspect is an elegant front end. It might serve as the basis of more complex analysis. Multi-pass algorithms are often *faster* and more elegant than single-pass algorithms, so the ``extra'' overhead of the leoInspect prepass is probably not significant. What matters are the deductions that can be made using leoInspect.Context data.

Speed is not an obstacle for a lint-like tool based on leoInspect. Indeed, the module getter creates *all* the context data in a very fast pass over the modules AST tree. It takes about 4.6 seconds to create the context data for all 34 of Leo's source files. Because getters are extremely fast, even complex queries will be fast. A multiple-pass query will typically take about 0.1 sec per pass. Using AstFormatter in the InspectTraverser class adds a negligible amount of time, less than 10% of the total tree-traversal time.

18.3 Still to do

The assignments_to, assignments_using and the new calls_to getters all specify a **pattern** to be matched against the the lhs of assignments (or against function names in the calls_to getter). At present, the pattern must match as a plain word match, but it would be more natural to use regex matches. That's coming.

Three new getters would give leoInspect the ability to replace refactored code:

- o.token_range: Returns pointers the list of tokens comprising o.
- o.text: Returns o's source text (a string).
- o.text_range: Returns the starting and ending offsets of the text in the file.

These getters are non-trivial to do, but a reasonable design is in place.

18.4 Theory of operation

The module getter creates *all* the data used by the other getters. This data is a richly-linked set of Context objects. Getters are very fast because the getter of an object o merely returns one of o's ivars.

The o.format getter is an exception. It traverses o's AST to create the human-readable representation of o.

The AstFormatter class is a straightforward recursive descent algorithm.

The InspectTraverser.do_Attribute method uses the *formatting* code to compute the value of the attribute. This replaces complex AST-traversal code with a call to the formatter.

18.4.1 Getters

Getters all follow roughly the same pattern. For example:

```
def assignments_to (self,s,all=True):

    format,result = self.formatter.format,[]

    for assn in self.assignments(all=all):
        tree = assn.tree()
        kind = self.ast_kind(tree)
        if kind == 'Assign':
            for target in tree.targets:
                lhs = format(target)
                if s == lhs:
                    result.append(assn)
                    break
        else:
            assert kind == 'AugAssign',kind
            lhs = format(tree.target)
            if s == lhs:
                result.append(assn)

    return result
```

This is AST-traversal code. Indeed, the tree getter returns an AST, and ast_kind is an internal getter that returns the AST type for an AST node.

This code is elegant. It uses the assignments getter to get the list of all assignments in this context and all contained (descendant) contexts. All public getters are members of the base Context class, so this code code ``just works" for *all* contexts. Furthermore, assignments are StatementContext objects, so they ``just work" as elements returned by the getter. Comparing this code with the code found in pylint shows how elegant this code truly is.

No simpler code is possible. AST trees for assignments are a bit different from AST trees for augmented assignments. Once the type of assignment is identified, the code simply *formats* the left hand side (lhs) of the assignment, and compares s with the lhs. If there is a match, the entire assignment is appended to the result.

18.4.2 Computing token ranges

This section describes the how the unfinished o.token_range getter will work.

There are two parts to the problem...

Token-info prepass

For each node N of a module's tree, we want to inject the following two new ivars:

- N.end_lineno: the line number of the last character of the token.
- N.end_col_offset: the (byte) offset of the last character of the token.

Important: tree structure is irrelevant when computing these fields: we simply want a **sorted** list of (N.lineno,No.col_offset, N) tuples!

The prepass will use ast.walk(root), to generate the list. After sorting the list, the prepass will inject inject N.end_lineno and N.end_col_offset ivars into each node N by stepping through the list. The ending values of the previous node on the list are the the same as the beginning values of the next node on the list.

This prepass need only be done once per module.

token_range

To compute token_range for a *particular* N, we want to discover values M.end_lineno and M.end_col_offset for M, the **last** token in N's entire tree.

token_range will do the prepass on the modules tree if necessary. token_range will then call ast.walk(N) to discover all of N's nodes, sort the list, and return the last element of the list!

token_range will, by its design, include *all* text in the range, including comments.

第 19 章

Leo and Emacs

This chapter several topics relating to the Emacs editor.

Contents

- Leo and Emacs
  - Controlling Leo from Emacs using Pymacs
  - Functions in leoPymacs.py
  - The minibuffer

19.1 Controlling Leo from Emacs using Pymacs

Leo's leoPymacs module is a simple `server' for the pymacs package. Using pymacs and leoPymacs, elisp scripts in Emacs can open .leo files and execute *Python* scripts as if they were executed inside Leo. In particular, such scripts can use Leo's predefined c, g and p variables. Thus, *Python* scripts running in Emacs can:

- Open any .leo file.
- Access any part of the outline.
- Change any part of the outline, including external files,
- Save .leo files.
- Execute *any* Leo script.

In short, you can now do from Emacs anything that you can do with Leo scripting inside Leo.

Here are step-by-step instructions for executing Python scripts in Emacs:

Step 1. Install pymacs

The pymacs installation instructions should be clear enough. A clarification is needed about two-way communication between Python and lisp scripts: in truth, Python scripts can call the Pymacs.lisp function *only* if the Python script was invoked from emacs. Otherwise, calling Pymacs.lisp will hang the process making the call. For example, executing the following script as an ordinary Leo script will hang Leo:

```
from Pymacs import lisp
print lisp("""2+2""") # Hangs
```

Step 2. Load the leoPymacs module from Emacs, creating a hidden Leo application

From inside Emacs, you load Leo's leoPymacs module as follows:

```
(pymacs-load "leoPymacs" "leo-")
```

The call to pymacs-load is similar to `import leoPymacs as leo-` in Python. The side effect of pymacs-load is to define the elisp function leo-x for every top-level function x in leoPymacs.py, namely leo-dump, leo-get-app, leo-get-g, leo-get-script-result, leo-init, leo-open and leo-run-script. The first call to any of these functions creates a **hidden Leo application** in which .leo files may be loaded, modified and saved, and in which Leo scripts may be executed. This hidden Leo application uses Leo's nullGui class as its gui, so Leo commands and Leo scripts that require a fully functional gui will not work as expected in the hidden Leo application. Steps 3 and 4 tell how to use this hidden Leo application.

pymacs-load works like a Python reload, so you can redefine leoPymacs.py while Emacs is running. However, calling pymacs-load destroys the old hidden Leo application and creates a new one, so typically you would want to call pymacs-load only once per Emacs session. Like this:

```
(setq reload nil) ; change nil to t to force a reload.

(if (or reload (not (boundp 'leoPymacs)))
   (setq leoPymacs (pymacs-load "leoPymacs" "leo-"))
   (message "leoPymacs already loaded")
)
```

Step 3. From Emacs, open .leo files

Once we have loaded the leoPymacs module we can open a .leo file as follows:

```
(setq c (leo-open fileName))
```

This binds the elisp c variable to the Leo commander created by opening fileName. fileName should be the full path to a .leo file. In the next step we will use this c variable to execute *Leo* scripts in the context of an open Leo outline.

Sometimes we want to execute a Leo script before opening any Leo commanders. For example, we might want to compute the fileName passed to leo-open. leo-run-script allows the c argument to be nil, in which case leo-run-script creates a dummy commander in which to run the script. For example, the following script calls g.os_path_join and g.os_path_abspath:

```
(setq script "g.app.scriptResult =
   g.os_path_abspath(g.os_path_join(
      g.app.loadDir,'..','test','ut.leo'))"
)
```

```
(setq fileName (leo-run-script nil script))
```

leo-run-script returns the value of g.app.scriptResult As shown above, Python scripts may set g.app.scriptResult to indicate their result. elisp scripts can also get g.app.scriptResult using leo-script-result. Note that the Python script may span multiple lines.

Step 4. From Emacs, execute Leo (Python) scripts

From emacs we can execute a Python script **as if** it were executed in an open Leo outline. Suppose aLeoScript is an **elisp** string containing a Leo (Python) script. We can execute that script in the hidden Leo application as follows:

```
(leo-run-script c aLeoScript)
```

For example:

```
(setq c (leo-open fileName)
(csetq script "print 'c',c,'h',c.p.h")
(leo-run-script c script)
```

Putting this all together, we get:

```
; Step 1: load leoPymacs if it has not already been loaded.
(setq reload nil)
(if (or reload (not (boundp 'leoPymacs)))
    (setq leoPymacs (pymacs-load "leoPymacs" "leo-"))
    (message "leoPymacs already loaded")
)

; Step 2: compute the path to leo/test/ut.leo using a Leo script.
(setq script
    "g.app.scriptResult = g.os_path_abspath(
        g.os_path_join(g.app.loadDir,'..','test','ut.leo'))"
)
(setq fileName (leo-run-script nil script))

; Step 3: execute a script in ut.leo.
(setq c (leo-open fileName))
(setq script "print 'c',c.shortFileName() ,'current:',c.p.h")
(leo-run-script c script)
```

19.2 Functions in leoPymacs.py

The leoPymacs module is intended to be called from Emacs using pymacs. It contains the following top-level functions:

- get_app()

  Returns the hidden app created by the leoPymacs.init function.

- dump(anyPythonObject)

Returns str(repr(anyPythonObject)).

- get_g()

  Returns the leoGlobals module of the hidden app created by the leoPymacs.init function.

- get_script_result()

  Returns g.app.scriptResult, where g.app is the hidden app.

- init() Calls leo.run(pymacs=True) to create a hidden Leo application. Later calls to open can open hidden Leo outlines that can be accessed via runScript.

- open(fileName)

  Opens the .leo file given by fileName. fileName must be the full path to a .leo file. Returns the commander of the open Leo outline, or None if the outline could not be opened.

- run_script(c,script,p=None)

  Executes a script in the context of a commander c returned by the leoPymacs.open. c may be None, in which case a dummy commander is created in which to run the script. In the executed script, p is set to c.p if no p argument is specified. Returns g.app.scriptResult, where g.app is the hidden app.

19.3 The minibuffer

Leo's mini-buffer is a text area at the bottom of the body pane. You use Leo's minibuffer like the Emacs mini-buffer to invoke commands by their so-called *long name*. The following commands affect the minibuffer:

- **full-command**: (default shortcut: Alt-x) Puts the focus in the minibuffer. Type a full command name, then hit <Return> to execute the command. Tab completion works, but not yet for file names.
- **quick-command-mode**: (default shortcut: Alt-x) Like Emacs Control-C. This mode is defined in leoSettings.leo. It is useful for commonly-used commands.
- **universal-argument**: (default shortcut: Alt-u) Like Emacs Ctrl-u. Adds a repeat count for later command. Ctrl-u 999 a adds 999 a's.
- **keyboard-quit**: (default shortcut: Ctrl-g) Exits any minibuffer mode and puts the focus in the body pane.

For example, to print a list of all commands type Alt-X print-commands <Return>.

第 **20** 章

IPython and Leo

Leo's ipython plugin provides two-way communication (a bridge) between Leo and IPython: you can run Leo scripts from IPython, and IPython scripts from Leo. To use this plugin, you must run Leo in a console window. When this plugin is enabled, Leo's start-ipython command starts IPython in this console.

Remarkably, Leo and IPython run simultaneously in the same process, yet their separate event loops do not interfere with each other. Scripts run from IPython *immediately* change Leo, *exactly* as if the script were run from Leo. Conversely, scripts run from Leo *immediately* affect the IPython interpreter. As a result, Leo might be considered an IPython Notebook.

The bridge between Leo and IPython is powerful because it is simple. Indeed,

1. **You can run any IPython script from Leo**. On the Leo side, a single statement:

```
ip = IPython.ipapi.get()
```

assigns ip to IPython's _ip variable. The ip variable allows scripts running in Leo to do *anything* that an IPython script can do.

2. **You can run any Leo script from IPython**. The ipython plugin injects a single object named `_leo' into the IPython namespace. IPython scripts access Leo's c and g objects as follows:

```
c,g = _leo.c, _leo.g
```

The c and g variables allow scripts running in IPython to do *anything* that a Leo script can do.

This is basically everything that is required for IPython-Leo interaction. However, you probably wont use `c' and `g' directly, but use a series of convenience wrappers described in this document that make interactive work painless and powerful.

Contents

- IPython and Leo
  - Introduction
  - Installation and startup
  - Accessing IPython from Leo
  - Accessing Leo nodes from IPython
  - @cl definitions
  - Special node types
  - Launching ILeo from IPython
  - Declaring custom push-to-ipython handlers
  - Example code snippets
  - Example use case: pylab
  - Magic functions
  - Acknowledgements and history

20.1 Introduction

ILeo, or leo-ipython bridge, creates a two-way communication channel between Leo and IPython. The level of integration is much deeper than conventional integration in IDEs; most notably, you are able to store and manipulate **data** in Leo nodes, in addition to mere program code - essentially making ILeo a hierarchical spreadsheet, albeit with non-grid view of the data. The possibilities of this are endless, and the approach can be applied in wide range of problem domains with very little actual coding.

IPython users are accustomed to using things like %edit to produce non-trivial functions/classes (i.e. something that they don't want to enter directly on the interactive prompt, but creating a proper script/module involves too much overhead). In ILeo, this task consists just going to the Leo window, creating a node and writing the code there, and pressing alt+I (push-to-ipython).

Obviously, you can save the Leo document as usual - this is a great advantage of ILeo over using %edit, you can save your experimental scripts all at one time, without having to organize them into script/module files (before you really want to, of course!)

20.2 Installation and startup

You need at least Leo 4.4.8, and IPython 0.8.3

The ILeo concept is still being developed actively, so if you want to get access to latest features you can get IPython from Launchpad by installing bzr and doing:

```
bzr branch lp:ipython
cd ipython
python setupegg.py develop
```

You need to enable the `ipython.py' plugin in Leo:

- Help -> Open LeoSettings.leo
- Edit @settings-->Plugins-->@enabled-plugins, add/uncomment `ipython.py'
- Alternatively, you can add @settings-->@enabled-plugins with body ipython.py to your leo document.
- Restart Leo. Be sure that you have the console window open (run Leo in a console window, or double-click leo.py on windows)
- When using the Qt ui, add --ipython argument to command line (e.g. launchLeo.py --ipython).
- Press alt+shift+i OR alt-x start-ipython to launch IPython in the console that started leo. You can start entering IPython commands normally, and Leo will keep running at the same time.
- Note that you can just press alt-I (push-to-ipython) - it will start IPython if it has not been previously started. However, when you open a new leo document, you have to execute start-ipython (alt+shift+I) again to tell IPython that the new commands should target the new document. IPython session will not be restarted, only the leo commander object is updated in the existing session.
- If you want to specify command line arguments to IPython (e.g. to choose a profile, or to start in `pylab' mode), add this to your @settings: `@string ipython_argv = ipython -pylab' (where -pylab is the command line argument)

20.3 Accessing IPython from Leo

20.3.1 IPython code

Just enter IPython commands on a Leo node and press alt-I to execute push-to-ipython in order to execute the script in IPython. `commands' is interpreted loosely here - you can enter function and class definitions, in addition to the things you would usually enter at IPython prompt - calculations, system commands etc.

Everything that would be legal to enter on IPython prompt is legal to execute from ILeo.

Results will be shows in Leo log window for convenience, in addition to the console.

Suppose that a node had the following contents:

```
1+2
print "hello"
3+4

def f(x):
    return x.upper()

f('hello world')
```

If you press alt+I on that node, you will see the following in Leo log window (IPython tab):

```
In: 1+2
<2> 3
In: 3+4
<4> 7
```

```
In: f('hello world')
<6> 'HELLO WORLD'
```

(numbers like <6> mean IPython output history indices; the actual object can be referenced with _6 as usual in IPython).

20.3.2 Plain Python code

If the headline of the node ends with capital P, alt-I will not run the code through IPython translation mechanism but use the direct python `exec' statement (in IPython user namespace) to execute the code. It wont be shown in IPython history, and sometimes it is safer (and more efficient) to execute things as plain Python statements. Large class definitions are good candidates for P nodes.

20.4 Accessing Leo nodes from IPython

The real fun starts when you start entering text to leo nodes, and are using that as data (input/output) for your IPython work.

Accessing Leo nodes happens through the variable **wb** (short for ``WorkBook") that exist in the IPython user namespace. Nodes that are directly accessible are the ones that have simple names which could also be Python variable names; `foo_1' will be accessible directly from IPython, whereas `my scripts' will not. If you want to access a node with arbitrary headline, add a child node `@a foo' (@a stands for `anchor'). Then, the parent of `@a foo' is accessible through `wb.foo'.

You can see what nodes are accessible be entering (in IPython) wb.<TAB>. Example:

```
[C:leo/core]|12> wb.
wb.b         wb.tempfile    wb.rfile       wb.NewHeadline
wb.bar       wb.Docs        wb.strlist     wb.csvr
[C:leo/core]|12> wb.tempfile
         <12> <ipy_leo.LeoNode object at 0x044B6D90>
```

So here, we meet the `LeoNode' class that is your key to manipulating Leo content from IPython!

20.4.1 LeoNode

Suppose that we had a node with headline `spam' and body:

```
['12',2222+32]
```

we can access it from IPython (or from scripts entered into other Leo nodes!) by doing:

```
C:leo/core]|19> wb.spam.v
        <19> ['12', 2254]
```

`v' attribute stands for `value', which means the node contents will be run through `eval' and everything you would be able to enter into IPython prompt will be converted to objects. This mechanism can be extended far beyond direct evaluation (see `@cl definitions').

`v' attribute also has a setter, i.e. you can do:

```
wb.spam.v = "mystring"
```

Which will result in the node `spam' having the following text:

```
'mystring'
```

What assignment to `v' does can be configured through generic functions (`simplegeneric' module, see ipy_leo.py for examples).

Besides v, you can set the body text directly through:

```
wb.spam.b = "some\nstring",
```

headline by:

```
wb.spam.h = 'new_headline'
```

(obviously you must access the node through wb.new_headline from that point onwards), and access the contents as string list (IPython SList) through `wb.spam.l'.

If you do `wb.foo.v = 12' when node named `foo' does not exist, the node titled `foo' will be automatically created and assigned body 12.

LeoNode also supports go() that focuses the node in the Leo window, and ipush() that simulates pressing alt+I on the node (beware of the possible recursion!).

You can access unknownAttributes by .uA property dictionary. Unknown attributes allow you to store arbitrary (pickleable) python objects in the Leo nodes; the attributes are stored when you save the .leo document, and recreated when you open the document again. The attributes are not visible anywhere, but can be used for domain-specific metadata. Example:

```
[C:leo/core]|12> wb.spam.uA['coords'] = (12,222)
[C:leo/core]|13> wb.spam.uA
         <13> {'coords': (12, 222)}
```

20.4.2 Accessing children with iteration and dict notation

Sometimes, you may want to treat a node as a `database', where the nodes children represent elements in the database. You can create a new child node for node `spam', with headline `foo bar' like this:

```
wb.spam['foo bar'] = "Hello"
```

And assign a new value for it by doing:

```
wb.spam['foo bar'].v = "Hello again"
```

Note how you can't use .v when you first create the node - i.e. the node needs to be initialized by simple assignment, that will be interpreted as assignment to `.v'. This is a conscious design choice.

If you try to do wb.spam['bar'] = `Hello', ILeo will assign `@k bar' as the headline for the child instead, because `bar' is a legal python name (and as such would be incorporated in the workbook namespace). This is done to avoid crowding the workbook namespace with extraneous items. The item will still be accessible as wb.spam['bar']

LeoNodes are iterable, so to see the headlines of all the children of `spam' do:

```
for n in wb.spam:
    print n.h
```

20.5 @cl definitions

If the first line in the body text is of the form `@cl sometext', IPython will evaluate `sometext' and call the result with the rest of the body when you do `wb.foo.v' or press alt+I on the node. An example is in place here. Suppose that we have defined a class (I use the term class in a non-python sense here):

```
def rfile(body,node):
    """ @cl rfile

    produces a StringIO (file like obj) of the rest of the body """

    import StringIO
    return StringIO.StringIO(body)
```

(note that node is ignored here - but it could be used to access headline, children etc.),

Now, let's say you have node `spam' with text:

```
@cl rfile
hello
world
and whatever
```

Now, in IPython, we can do this:

```
[C:leo/core]|22> f = wb.spam.v
[C:leo/core]|23> f
         <23> <StringIO.StringIO instance at 0x04E7E490>
[C:leo/core]|24> f.readline()
         <24> u'hello\n'
[C:leo/core]|25> f.readline()
         <25> u'world\n'
[C:leo/core]|26> f.readline()
         <26> u'and whatever'
[C:leo/core]|27> f.readline()
         <27> u''
```

You should declare new @cl types to make ILeo as convenient your problem domain as possible. For example, a ``@cl etree'' could return the elementtree object for xml content.

In the preceding examples, the return value matter. That, of course, is optional. You can just use the @cl node as a convenient syntax for ``run this body text through a function''.

Consider this example:

```
def remote(body, node):
    out = sshdo(body)
    c = node.append()
    c.b = "@nocolor\n" + out
    c.h = "Command output"
```

(sshdo(s) is a just a trivial function implemented using paramiko, that returns the output from command run over ssh on remote host).

After running the above node (by, say, wb.require(`remote_impl') if the function is declared in a node named `remote_impl'), you can create nodes that have various little sysadmin tasks (grep the logs, gather data, kick out all the users) like this:

```
@cl remote
cd /var/log
ls -l
echo " --- temp ---"
cd /var/tmp
ls -l
```

Press alt+I on the node to run it. The output will be written to ``Command output" child node.

20.6 Special node types

20.6.1 @ipy-startup

If this node exist, the *direct children* of this will be pushed to IPython when ILeo is started (you press alt+shift-i). Use it to push your own @cl definitions, import the modules you will be using elsewhere in the document, etc.

The contents of of the node itself will be ignored.

20.6.2 @ipy-results

If you press alt+I on a node that has @cl, it will be evaluated and the result will be put into this node. Otherwise, it will just be displayed in log tab.

20.6.3 @ipy-root

You can set up a subportion of the leo document as a ``sandbox" for your IPython work. Only the nodes under @ipy-root will be visible through the `wb' variable.

Also, when you create a new node (wb.foo.v = `stuff'), the node foo will be created as a child of this node.

20.6.4 @a nodes

You can attach these as children of existing nodes to provide a way to access nodes with arbitrary headlines, or to provide aliases to other nodes. If multiple @a nodes are attached as children of a node, all the names can be used to access the same object.

20.7 Launching ILeo from IPython

Sometimes you may decide to launch Leo when an IPython session is already running. This is typically the case when IPython is launched from/as another application (Turbogears/Django shell, Sage, etc.), or you only decide later on that you might want to roll up some scripts or edit your variables in Leo.

Luckily, this is quite easy, if not automatic (yet) using IPython's %run command that runs python code in the IPython process. The only special consideration is that we need to run IPython.Shell.hijack_tk() to prevent Leo Tk mainloop from blocking IPython in %run. Here we launch an embedded Leo instance, and create a macro `embleo' for later use (so that we don't have to repeat these steps):

```
IPython 0.8.3.bzr.r57   [on Py 2.5.1]
[C:opt/Console2]|2> import IPython.Shell
[C:opt/Console2]|3> IPython.Shell.hijack_tk()
[C:opt/Console2]|4> cd c:/leo.repo/trunk
[c:leo/leo.repo/trunk]|5> %run launchLeo.py

reading settings in C:\leo\leo\config\leoSettings.leo

... Leo is starting at this point, but IPython prompt returns ...

[c:leo/leo.repo/trunk]|6> macro embleo 2-5

[c:leo/leo.repo/trunk]|7> store embleo
Stored 'embleo' (Macro)
```

Now, in Leo, you only need to press Alt+Shift+I (launch-ipython) to actually make the document visible in IPython. Despite the name, launch-ipython will not create a new instance of IPython; if an IPython session already exists, it will be automatically used by ILeo.

20.8 Declaring custom push-to-ipython handlers

Sometimes, you might want to configure what alt+I on a node does. You can do that by creating your own push function and expose it using ipy_leo.expose_ileo_push(f, priority). The function should check whether the node should by handled by the function and raise IPython.ipapi.TryNext if it will not do the handling, giving the next function in the chain a chance to see whether it should handle the push.

This example would print an uppercase version of node body if the node headline ends with U (yes, this is completely useless!):

```
def push_upcase(node):
    if not node.h.endswith('U'):
        raise TryNext
    print node.b.upper()

ipy_leo.expose_ileo_push(push_upcase, 12)
```

(the priority should be between 0-100, with 0 being the highest (first one to try) - typically, you don't need to care about it and can usually omit the argument altogether)

20.9 Example code snippets

Get list of all headlines of all the nodes in leo:

```
[node.h for node in wb]
```

Create node with headline `baz', empty body:

```
wb.baz
```

Create 10 child nodes for baz, where i is headline and `Hello ` + i is body:

```
for i in range(10):
    wb.baz[i] = 'Hello %d' % i
```

Create 5 child nodes for the current node (note the use of special _p variable, which means ``current node'') and moves focus to node number 5:

```
for i in range(10):
    _p[i] = 'hello %d' % d
_p[5].go()
```

Sort contents of a node in alphabetical order (after pushing this to IPython, you can sort a node `foo' in-place by doing sort_node(wb.foo)):

```
def sort_node(n):
    lines = n.l
    lines.sort()
    n.l = lines
```

20.10 Example use case: pylab

If you install matplotlib and numpy, you can use ILeo to interactively edit and view your data. This is convenient for storing potentially valuable information in Leo document, and yields an interactive system that is comparable in convenience to various commercial mathematical packages (at least if you compare it against plain IPython, that forgets the data on exit unless explicitly saved to data files or %store:d).

20.10.1 Startup

It's probably safest to rely on TkAgg back end, to avoid two event loops running in the same process. TkAgg is the default, so the only thing you need to do is to install numpy and matplotlib:

```
easy_install numpy
easy_install matplotlib
```

Finally, you need to start up IPython with `-pylab' option. You can accomplish this by having the following under some @settings node:

```
@string ipython_argv = ipython -pylab
```

Then, you just need to press alt+I to launch IPython.

20.10.2 Usage

The simplest use case is probably pushing an existing array to Leo for editing. Let's generate a simple array and edit it:

```
[c:/ipython]|51> a = arange(12).reshape(3,4)
[c:/ipython]|52> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
[c:/ipython]|53> %lee a
```

This (the magic command %lee, or `leo edit') will open variable `a' for editing in Leo, in a convenient pretty-printed format. You can press alt+I on the node to push it back to IPython.

If you want to store the variable in a node with a different name (myarr), you can do:

```
[c:/ipython]|54> wb.myarr.v = a
```

Then, you can always get the value of this array with wb.myarr.v. E.g. you could have a node that plots the array, with content:

```
# press alt+i here to plot testarr

plot(wb.myarr.v)
```

And, as per instructions, pressing alt+I will launch a new Tk window with the plotted representation of the array!

20.11 Magic functions

20.11.1 %mb

Execute leo minibuffer command. Tab completion works. Example:

```
mb open-outline
```

20.11.2 %lee

Stands for ``LEo Edit''. Allows you to open file(s), and even objects in Leo for editing. Examples:

```
lee *.txt
```

Opens all txt files in @auto nodes

```
lee MyMacro
```

Opens the macro MyMacro for editing. Press alt-I to push the edited macro back to IPython.

```
s = 'hello word'
lee s
```

Opens the variable s for editing. Press alt+I to push the new value to IPython.

```
lee hist
```

Opens IPython interactive history (both input and output) in Leo.

20.12 Acknowledgements and history

This idea got started when I (Ville M. Vainio) saw this post by Edward Ream (the author of Leo) on IPython developer mailing list:

> http://lists.ipython.scipy.org/pipermail/ipython-dev/2008-January/003551.html

I was using FreeMind as mind mapping software, and so I had an immediate use case for Leo (which, incidentally, is superior to FreeMind as mind mapper). The wheels started rolling, I got obsessed with the power of this concept (everything clicked together), and Edwards excitement paralleled mine. Everything was mind-bogglingly easy/trivial, something that is typical of all promising technologies.

The goal of close cooperation between Leo and IPython went from vague dream to completed reality over the span of about 10 days.

第 **21** 章

Embedding Leo with the leoBridge module

The leoBridge module allows complete access to all aspects of Leo from other Python programs running independently of Leo. Let us call such a program a **host** program. Using the leoBridge module, host programs can get access to:

- all of Leo's source code,
- the contents of any .leo file,
- the commander of any .leo file.

Contents

- Embedding Leo with the leoBridge module
  - The basics
  - Running leoBridge from within Leo

21.1 The basics

Host programs use the leoBridge module as follows:

```
import leo.core.leoBridge as leoBridge

controller = leoBridge.controller(gui='nullGui',
    loadPlugins=True,  # True: attempt to load plugins.
    readSettings=True, # True: read standard settings files.
    silent=False,      # True: don't print signon messages.
    verbose=False)     # True: print informational messages.

g = controller.globals()
c = controller.openLeoFile(path)
```

Let us look at these statements in detail. The first two statements import the leoBridge module and create a **bridge controller**. In effect, these statements embed an invisible copy of Leo into the host program. This embedded copy of Leo uses a null gui, which simulates all aspects of Leo's normal gui code without creating any screen objects.

The statement:

```
g = controller.globals()
```

provides access to Leo's leoGlobals module, and properly inits globals such as g.app, g.app.gui, etc. *Host programs should not import leoGlobals directly*, because doing so would not init the g.app object properly.

The statement:

```
c = controller.openLeoFile(path)
```

invisibly opens the .leo file given by the path argument. This call returns a completely standard Leo commander, properly inited. This is the big payoff from the leoBridge module: the host program gets instant access to c.config.getBool, etc. Do you see how sweet this is?

For example, the following script runs leo/test/leoBridgeTest.py outside of Leo. leoBridgeTest.py uses the leoBridge module to run all unit tests in leo/test/unitTest.leo:

```
import os,sys

path = g.os_path_abspath(
    g.os_path_join(
        g.app.loadDir,'..','test','leoBridgeTest.py'))

os.system('%s %s' % (sys.executable,path))
```

The file leo/test/test.leo contains the source code for leoBridgeTest.py. Here it is, stripped of its sentinel lines:

```
'''A program to run unit tests with the leoBridge module.'''

import leo.core.leoBridge as leoBridge
import leo.core.leoTest as leoTest

def main ():
    tag = 'leoTestBridge'

    # Setting verbose=True prints messages that would be sent to the log pane.
    bridge = leoBridge.controller(gui='nullGui',verbose=False)
    if bridge.isOpen():
        g = bridge.globals()
        path = g.os_path_abspath(g.os_path_join(
            g.app.loadDir,'..','test','unitTest.leo'))
        c = bridge.openLeoFile(path)
        g.es('%s %s' % (tag,c.shortFileName()))
        runUnitTests(c,g)

    print tag,'done'
```

```
def runUnitTests (c,g):
    nodeName = 'All unit tests' # The tests to run.
    try:
        u = leoTest.testUtils(c)
        p = u.findNodeAnywhere(nodeName)
        if p:
            g.es('running unit tests in %s...' % nodeName)
            c.selectPosition(p)
            c.debugCommands.runUnitTests()
            g.es('unit tests complete')
        else:
            g.es('node not found:' % nodeName)
    except Exception:
        g.es('unexpected exception')
        g.es_exception()
        raise

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

21.2 Running leoBridge from within Leo

This following is adapted from Terry Brown's entry in Leo's wiki.

You can not just run leoBridge from Leo, because the leoBridge module is designed to run a separate copy of Leo. However, it is possible to run leoBridge from a separate process. That turned out to be more, um, interesting than anticipated, so I'm recording the results here.

The idea is that script A running in Leo (i.e. in a regular GUI Leo session) calls script B through subprocess.Popen(), script B uses LeoBridge to do something (parse unloaded Leo files), and returns the result to script A. Passing the result back via the clipboard seemed like a possibility, but XWindows / tcl/tk clipboard madness being what it is, that didn't seem to work.

First trick, calling script B from script A:

```
import subprocess
p = subprocess.Popen(('python',
    path_to_script_B,
    parameter_for_script_B,),
    stdout=subprocess.PIPE,
    env={'PYTHONPATH': g.app.loadDir,'USER': g.app.leoID},
)
p.wait()
```

Setting PYTHONPATH in the environment seemed like the easiest way to let script B find leoBridge.py (which it needs to import). But by setting the env parameter you limit script B's environment to be **only** PYTHONPATH, which causes leoBridge to fail because, in unix at least, it depends on USER in the environment. So you need to pass that through, too.

Now, because passing stuff back on the clipboard seems unreliable, at least in XWindows, script B passes results back to script A via stdout (print), but there's some Leo initialization chatter you want to avoid. So put a sentinel, `START_CLIPBOARD', in the output, and collect it like this:

```
response = p.stdout.readlines()
while response and 'START_CLIPBOARD' not in response[0]:
    del response[0]
del response[0]  # delete the sentinel as well
response = ''.join(response)
```

This is the basic mechanism. What I *actually* wanted to do was have script B generate a branch of nodes and pass that back to script A for insertion in the tree script A is running in. That's relatively easy if you use:

```
c.setCurrentPosition(pos_of_branch_to_return)
c.copyOutline()
print '<!-- START_CLIPBOARD -->'
print g.app.gui.getTextFromClipboard()
print '<!-- END_CLIPBOARD -->'
```

at the end of script B. Back in script A, after you've rebuilt *response* as shown above, do:

```
g.app.gui.replaceClipboardWith(response)
c.pasteOutline()
```

第 **22** 章

Using Vim Bindings with Leo

This chapter describes Leo's vim-like bindings, including how to install them.

Contents

- Using Vim Bindings with Leo
  - Installing vim bindings
  - General commands
  - Body pane commands
  - Outline commands
  - Commands not supported
  - Avoiding changes to `tag' files

22.1 Installing vim bindings

Place a copy of the ``@keys Vim bindings" node and its sub-nodes, located in the leoSettings.leo file, under the ``@settings" node in the myLeoSettings.leo file

The same procedure is performed to update to a new version.

Note: Place any local customized key bindings in a separate ``@keys My Vi" node in the myLeoSettings.leo file to prevent them from being overwritten when updating to a new version.

22.2 General commands

The following commands are always available.

State change commands:

```
i         Change state to insert from command state
Esc       Change state to command from insert state
Ctrl-[    Same as ESC
```

Save/Exit/Quite commands:

```
:e          Revert
:w<return>  Save '.leo' file
:wq<return> Save '.leo' file and quit Leo
:q<return>  Quit Leo   (Leo will prompt if file not saved)
ZZ          Save leo file and exit
```

Undo/Redo commands:

```
u           Undo previous command
Ctrl-r      Redo previous command
```

Search options:

```
Ctrl-/      Prompt for option to change
         Options:
           a   Search all nodes (also <cr> key)
           h   Toggle headline search
           b   Toggle body search
           m   Toggle marking of nodes (specify sub-option)
              f   Toggle marking of nodes with found text
              c   Toggle marking of nodes with changed text
                 (only supported with 'Alt-/', 'Alt-p')
           r   Toggle regex matches
              ('/' key turns off regex. 'n' key uses regex if turned on)

Note: Whether a search is limited to node's body or the node's sub-outline
    is determined by which pane has focus when search text specified.
    (See "Find text commands:" sub-sections in Outline/Body Pane sections)
```

Miscellaneous commands:

```
Tab         Toggle focus between Outline and Body pane
=           Simulate double-click on current node's icon box
Alt-G       Go to specified line number (relative to external file)
Ctrl-:      Enter Leo's command line
```

22.3 Body pane commands

Move cursor commands:

```
h           Go back 1 character
  LtArrow  Mapped to "h" for convenience
j           Go down 1 line
  DnArrow  Mapped to "j" for convenience
```

```
k          Go up 1 line
  UpArrow  Mapped to "k" for convenience
l          Go forward 1 character
  RtArrow  Mapped to "l" for convenience

w          Go to beginning of next word
  W        Mapped to "w" until "stop after blank characters" supported
b          Go to beginning of current/previous word
  B        Mapped to "b" until "stop at blank character" supported
e          Go to end of current/next word
  E        Mapped to "e" until "stop at blank character" supported

Note: Move by word commands stop at non-alpha characters

|          Goto beginning of current line
^          Go to 1st non-blank character on current line
$          Goto end of current line

%          Go to matching bracket

(          Go to beginning of current sentence
)          Go to beginning of next sentence
{          Go to beginning of current paragraph
}          Go to beginning of next paragraph

gg         Go to the first line (Cursor at column 1)
G          Go to the last line  (Cursor at column 1)
```

Mark commands:

```
m<label>   Assign cursor location to a single character label
`<label>   Go to location associated with label

 Note: Only character count is tracked. Any inserts or deletes will change mark.
    Mark's are not node specific; `<label> will go to location in current node.
```

Select commands:

```
Ctrl-v     Toggle text select mode (Vim's "visual" mode)
  V        Mapped to 'Ctrl-v' for convenience (Should toggle line select)
```

Insert/substitute commands:

```
a          Insert at cursor
i          Mapped to "a" until "cursor on a character" supported
A          Insert at end of line
I          Insert at first non-space
o          Open new line below current line
O          Open new line above current line
R          Overwrite text
```

s	Substitute character (Delete character, enter insert state)
S	Substitute line (Delete line, enter insert state)

Change commands:

C	Change to end of line
cc	Change all of current line
cw	Change to end of word
cb	Change to beginning of word
c)	Delete to end of sentence
c(	Delete to beginning of sentence
c}	Delete to end of paragraph
c{	Delete to beginning of paragraph
c%	Change from current bracket type its matching bracket type
ct<char>	Selects forward to <char> (follow with 'i' to change selection)
cT<char>	Selects backward to <char> (follow with 'i' to change selection)
c<cr>	Change selected text

Delete commands:

x	Delete next character
delete	Delete next character
D	Delete to the end of the current line
dd	Delete current line
dw	Delete to end of word
db	Delete to beginning of word
d)	Delete to end of sentence
d(	Delete to beginning of sentence
d}	Delete to end of paragraph
d{	Delete to start of paragraph
d%	Delete from current bracket type to its apposing bracket
dt<ch>	Delete to character (not limited to current line)
d<cr>	Delete selected text
J	Join next line to end of current line (deletes carriage return)

Yank text commands:

Y	Yank to end of line
yy	Yank line
yw	Yank to beginning of next word
yb	Yank to beginning of current word
y)	Yank to end of sentence
y(	Yank to beginning of sentence
y}	Yank to end of paragraph
y{	Yank to beginning of paragraph
y%	Yank from current bracket type to its opposing bracket
yt<char>	Select forward to <char> (use 'y<cr>' to yank selection)
yT<char>	Select backward to <char> (use 'y<cr>' to yank selection)
y<cr>	Yank selected text (Vim uses 'y' in visual mode)

Find character commands:

```
f          Find next occurrence of user specified character
F          Find previous occurrence of user specified character
```

Find text commands:

```
/          Search forward within current node's body text
?          Search backward within current node's body text
n          Find next (same scope, same direction)
N          Find next (same scope, other direction)

Note: See "Search options" in General Commands section to change options.
```

Replace [and find next] commands:

```
Commands using Paste buffer (clipboard)
P          Paste text before cursor.
p          Mapped to "P" until character based cursor supported.
Ctrl-p     Paste then find next match
           Note: Use 'pn' instead of 'Ctrl-p' in headlines (Leo limitation)
                 Command will continue to paste when match no longer found.

Commands prompting for replace string
Note: Scope and direction taken from last use of '/','?' or 'Ctrl-/'(scope only)
Alt-/      Prompt for search & replace string
Alt-p      Replace then search (use after Alt-/)
           Note: Works in headlines and body panes.
                 Doesn't paste unless last search found a match.
```

Indent/Unindent line commands:

```
>>         Indent the current line
>)         Indent to the end of sentence
>(         Indent to the beginning of sentence
>}         Indent to the end of paragraph
>{         Indent to the beginning of paragraph
>g         Indent to the start of buffer
>G         Indent to the end of buffer

<>         Unindent the current line
<)         Unindent to the end of sentence
<(         Unindent to the beginning of sentence
<}         Unindent to the end of paragraph
<{         Unindent to the beginning of paragraph
<g         Unindent to the start of buffer
<G         Unindent to the end of buffer
```

Format commands:

```
gqap       Split long line into separate lines
gwap       Split long line into separate lines
gqq        Split long line into separate lines
gww        Split long line into separate lines

Note: 'gwap' and 'gww' should not move cursor but do.
```

Scroll commands:

```
Ctrl-b      Scroll text up by pane's height
Ctrl-f      Scroll text down by pane's height
  Ctrl-y    Mapped to Ctrl-b until scroll up one line is supported
  Ctrl-e    Mapped to Ctrl-f until scroll down one line is supported
  Ctrl-u    Mapped to Ctrl-b until scroll up half a pane height is supported
  Ctrl-d    Mapped to Ctrl-f until scroll down half a pane height is supported
```

Window commands:

```
Ctrl-w s    Open another view into current node's body (Vim: Split window)
  Ctrl-w n Mapped to "Ctrl-w s" (Vim: New buffer in split window)
Ctrl-w w    Switch to next view (Vim: Go to up/left window w/wrapping)
  Ctrl-w p Mapped to "Ctrl-w w" (Vim: Cycle through windows)
  Ctrl-w k Mapped to "Ctrl-w w" (Vim: Go to window above current window)
  Ctrl-w j Mapped to "Ctrl-w w" (Vim: Go to window below current window)
Ctrl-w c    Close current view in body pane (Vim: Close current window)
  Ctrl-w q Mapped to "Ctrl-w c" (Vim: Quit current window)
```

Node commands:

```
Go to another node while focus remains in the body pane.
Ctrl-j      Go to next visible node
Ctrl-k      Go to previous visible node
Ctrl-h      Hide sub-nodes or, if hidden, go up 1 level
Ctrl-l      Display sub-nodes or, if displayed, go down 1 level
  Ctrl-DnArrow    Mapped to "Ctrl-j" for convenience
  Ctrl-UpArrow    Mapped to "Ctrl-k" for convenience
  Ctrl-LtArrow    Mapped to "Ctrl-h" for convenience
  Ctrl-RtArrow    Mapped to "Ctrl-l" for convenience
```

22.4 Outline commands

The following commands are supported when in a headline's command mode.

State change commands:

```
Ctrl-i      Change state to command from grayed state
return      Change state to command from insert state
Ctrl-]      Change state to grayed from command state
```

Cursor movement commands:

```
h          Go to previous character
  LtArrow  Mapped to 'h' for convenience
l          Go to next character
  RtArrow  Mapped to "l" for convenience

Note: 'j' and 'k' will scroll the buffer contents up and down;
     leaving the focus in the outline pane.

w          Go to beginning of next word
  W        Mapped to "w" until "stop after blank characters" supported
b          Go to beginning of current/previous word
  B        Mapped to "b" until "stop at blank character" supported
e          Go to end of current/next word
  E        Mapped to "e" until "stop at blank character" supported

Note: Move by word commands stop at non-alpha characters

|          Go to beginning of line
^          Go to beginning of line
$          Go to end of line

%          Go to matching bracket
```

Edit commands:

```
x          Delete next character
delete     Delete next character
dd         kill-line

s          Select current character

v          Toggle text select mode (issue cursor movement commands)
y<return>  Yank selected text

C          Select to end of line (follow with 'i' to change text)
cc         Delete line (follow with 'i' to change text)

D          Select to end of line (follow with 'x' to delete text)
dd         Delete line

Y          Select to end of line (follow with 'y<return>' to yank text)
yy         Select line (follow with 'y<return>' to yank text)
```

Find character commands:

```
f          Find next occurrence of user specified character
F          Find previous occurrence of user specified character
```

Find text commands:

```
/         Search forward within current node and its subnodes
n          Find next (same scope, same direction)
N          Find next (same scope, other direction)

Note: See "Search options" section above to change options using 'Ctrl-/'
```

Replace [and find next] commands:

```
Commands that use Paste buffer (clipboard)
Note: Paste-then-search command not possible in headlines (Use 'pn')
P          Paste text before cursor.
p          Mapped to "P" until character based cursor supported.

Commands that prompt for the replace string
Alt-/      Prompt for search & replace string
Alt-p      Replace then search (use after Alt-/)
          Note: Works in headlines and body panes.
               Doesn't paste unless last search found a match.
```

Node edit commands:

```
o          Insert node after current node

Ctrl-x     Delete current node
Ctrl-c     Yank current node
Ctrl-v     Paste current node
```

Node goto commands:

```
G          Go to the outline's last node
gg         Go to the outline's first node

Ctrl-j     Go to next visible node
Ctrl-k     Go to previous visible node
Ctrl-h     Hide sub-nodes or, if hidden, go up 1 level
Ctrl-l     Display sub-nodes or, if displayed, go down 1 level

  DnArrow  Mapped to "Ctrl-j" for convenience
  UpArrow  Mapped to "Ctrl-k" for convenience

  Ctrl-DnArrow Mapped to "Ctrl-j" for convenience
  Ctrl-UpArrow Mapped to "Ctrl-k" for convenience
  Ctrl-LtArrow Mapped to "Ctrl-h" for convenience
  Ctrl-RtArrow Mapped to "Ctrl-l" for convenience
```

Node move commands:

```
Ctrl-Shift-k   Move node down
Ctrl-Shift-h   Move node left
Ctrl-Shift-l   Move node right
Ctrl-Shift-j   Move node up
```

```
  Ctrl-Shift-DnArrow    Mapped to "Ctrl-Shift-k" for convenience
  Ctrl-Shift-LtArrow    Mapped to "Ctrl-Shift-h" for convenience
  Ctrl-Shift-RtArrow    Mapped to "Ctrl-Shift-l" for convenience
  Ctrl-Shift-UpArrow    Mapped to "Ctrl-Shift-j" for convenience
```

Node mark commands:

```
m         Toggle node mark
Ctrl-m    Go to next marked node
Alt-m     Clear all marked nodes
```

Node clone commands:

```
t         Clone the current node (transclude)
Ctrl-t    Go to next clone of current node
```

Outline scroll commands:

```
Ctrl-y    Scroll outline up one line
Ctrl-e    scroll outline down one line
Ctrl-u    Scroll outline up one half page
Ctrl-d    scroll outline down one half page
Ctrl-b    Scroll outline up one page
Ctrl-f    scroll outline down one page
```

22.5 Commands not supported

Notable missing editing commands:

```
t<char>   Move cursor to character before specified character
r         Replace a single character with a single character
0         Go to 1st column in current line (Use '|' instead)
bksp      Move one character to the left
~         Toggle character's case
.         Repeat last editing command
;         Repeat last cursor movement command
<n><cmd>  Perform command 'n' number of times
<cmd><n><object>  Perform the command on the n'th or up to the n'th object
```

Notable missing body pane commands:

```
<num>G    Go to specified line number
z<movement> Slide buffer to put current line at top/middle/bottom of pane
'<command>  Go to line of last edit, jump, ...
`<command>  Go to character of last edit, jump, ...
```

22.6 Avoiding changes to `tag' files

If you use the open-with plugin to open node text in Vim and your Vim's ``tag" file refers to external files then there is a risk that a external file that is initially displayed via the ``tag" command" in Vim is accidentally edited and saved from the external Vim editor while your Leo session still contains the external file's original text that may later recreate the original external file during a Leo save operation (overwriting the changes saved from the Vim editor).

To prevent this problem, modifications to external files can be avoided by using Vim's ``modeline" feature to disable editing of external files.

Vim's ``modeline" feature scans each loaded buffer for text at the top or bottom of the file containing " vim:" followed by a series of Vim options. The text is usually embedded within a comment. The following example prevents modifications to a buffer in a Python file:

```
# vim:noma (A space is required between the '#' and "vim:noma")
```

If this line is placed in a separate Leo node at the top or bottom of the list of nodes under a external file node (ex: @file) then any external file saved and then later loaded into Vim will, by default, not be modifiable. If a derived file does need to be edited then modifications can be re-enabled on a file-by-file basis by issuing Vim's ":ma" command while viewing the derived file.

The number of lines that Vim checks at the top and bottom of the buffer is configurable. The following Vim command must be placed in the vimrc file to allow for Leo's trailing sentinel lines:

```
set modelines=8
```

Issue the ":help modeline" command within Vim for the more information about modelines.

第 **23** 章

Using ZODB with Leo

This chapter discusses how to write Leo scripts that store and retrieve data using ZODB.

Contents

- Using ZODB with Leo
  - Configuring Leo to use zodb
  - Initing zodb
  - Writing data to zodb
  - Defining zodb keys
  - Reading data from zodb
  - About connections
  - Convenience routines

23.1 Configuring Leo to use zodb

To enable zodb scripting within Leo, you must set use_zodb = True in the root node of leoNodes.py. You must also install ZODB itself. See Installing ZODB for details.

When ZODB is installed and use_zodb is True, Leo's vnode class becomes a subclass of ZODB.Persistence.Persistent. This is all that is needed to save/retrieve vnodes or tnodes to/from the ZODB.

Important notes:

- Scripts **should not** store or retrieve positions using the ZODB! Doing so makes sense neither from Leo's point of view nor from ZODB's point of view.
- The examples below show how to store or retrieve Leo data by accessing the so-called root of a ZODB connection. However, these are only examples. Scripts are free to do with Leo's vnodes *anything* that can be done with ZODB.Persistence.Persistent objects.

23.2 Initing zodb

Scripts should call g.init_zodb to open a ZODB.Storage file. g.init_zodb returns an instance of ZODB.DB. For example:

```
db = g.init_zodb (zodbStorageFileName)
```

You can call g.init_zodb as many times as you like. Only the first call for any path actually does anything: subsequent calls for a previously opened path simply return the same value as the first call.

23.3 Writing data to zodb

The following script writes v, a tree of vnodes, to zodb:

```
db = g.init_zodb (zodbStorageFileName)
connection = db.open()
try:
    root = connection.root()
    root[aKey] = v # See next section for how to define aKey.
finally:
    get_transaction().commit()
    connection.close()
```

Notes:

- v must be a vnode. Scripts should *not* attempt to store Leo positions in the zodb. v can be the root of an entire outline or a subtree. For example, either of the following would be reasonable:

  ```
  root[aKey] = c.rootPosition().v
  root[aKey] = c.p.v
  ```

- To write a single vnode without writing any of its children you can use v.detach. For example:

  ```
  root[aKey] = v.detach()
  ```

- **Important**: It is simplest if only one zodb connection is open at any one time, so scripts would typically close the zodb connection immediately after processing the data. The correct way to do this is in a finally statement, as shown above.
- The script above does not define aKey. The following section discusses how to define reasonable zodb keys.

23.4 Defining zodb keys

The keys used to store and retrieve data in connection.root() can be any string that uniquely identifies the data. The following are only suggestions; you are free to use any string you like.

1. When saving a file, you would probably use a key that is similar to a real file path. For example:

   ```
   aKey = c.fileName()
   ```

2. When saving a single vnode or tree of vnodes, say v, a good choice would be to use v's gnx, namely:

   ```
   aKey = g.app.nodeIndices.toString(v.fileIndex)
   ```

   Note that v.detach() does not automatically copy v.fileIndex to the detached node, so when writing a detached node you would typically do the following:

   ```
   v2 = v.detach()
   v2.fileIndex = v.fileIndex
   aKey = g.app.nodeIndices.toString(v2.fileIndex)
   ```

23.5 Reading data from zodb

The following script reads a tree of vnodes from zodb and sets p as the root position of the tree:

```
try:
    connection = db.open()
    root = connection.root()
    v = root.get(aKey)
    p = leoNodes.position(v)
finally:
    get_transaction().commit()
    connection.close()
```

23.6 About connections

The scripts shown above close the zodb connection after processing the data. This is by far the simplest strategy. I recommend it for typical scripts.

Important: you must **leave the connection open** if your script modifies persistent data in any way. (Actually, this statement is not really true, but you must define zodb transaction managers if you intend to use multiple connections simultaneously. This complication is beyond the scope of this documentation.) For example, it would be possible to create a new Leo outline from the data just read, but the script must leave the connection open. I do not recommend this tactic, but for the adventurous here is some sample code:

```
connection = self.db.open()
root = connection.root()
v = root.get(fileName)
if v:
    c2 = c.new()
    c2.openDirectory = c.openDirectory # A hack.
    c2.mFileName = fileName # Another hack.
    c2.beginUpdate()
    try:
        c2.setRootVnode(v)
        c2Root = c2.rootPosition()
        c2.atFileCommands.readAll(c2Root)
```

```
        g.es_print('zodb read: %s' % (fileName))
    finally:
        c2.endUpdate()
    # Do *not* close the connection while the new Leo window is open!
else:
    g.es_print('zodb read: not found: %s' % (fileName))
```

This will work **provided** that no other zodb connection is ever opened while this connection is opened. Unless special zodb precautions are taken (like defining zodb transaction managers) calling get_transaction().commit() will affect **all** open connections. You have been warned.

23.7 Convenience routines

23.7.1 g.init_zodb (pathToZodbStorage,verbose=True)

This function inits the zodb. pathToZodbStorage is the full path to the zodb storage file. You can call g.init_zodb as many times as you like. Only the first call for any path actually does anything: subsequent calls for a previously opened path simply return the same value as the first call.

23.7.2 v.detach()

This vnode method returns v2, a copy of v that is completely detached from the outline. v2.fileIndex is unrelated to v.fileIndex initially, but it may be convenient to copy this field:

```
v2 = v.detach()
v2.fileIndex = v.fileIndex
```

第 **24** 章

Leo's Reference

This chapter is a reference for all of Leo's directives.

This chapter does *not* teach how to use Leo. It assumes you are *thoroughly* familiar with Leo's tutorial.

Contents

- Leo's Reference
  - Part 1: @<file> directives
    - @asis <path>
    - @auto <path>
    - @edit <path>
    - @file <path> (aka @thin)
    - @nosent <path>
    - @shadow <path>
  - Part 2: @all and @others
  - Part 3: Syntax coloring directives
  - Part 4: Dangerous directives
  - Part 5: All other directives

24.1 Part 1: @<file> directives

This section discusses the @<file> directives. These directives create or import external files.

Important: Newcomers to Leo should create external files with @auto or @file. Use @auto if your external files must not contain sentinel lines. Use @file otherwise. In particular, @file is **highly recommended** when sharing external files in a collaborative environment.

Note: All these directive must appear in headlines.

The following table summarizes the various ways of creating external files.

Kind	Sentinels in external file?	Sections and @others expanded on write?	File data stored in .leo file?
@asis	no	no	yes
@auto	no	yes	no
@edit	no	yes: note 3	no
@nosent	no	yes	yes
@shadow	Note 1	yes	no
@file @thin Note 2	yes	yes	no

Note 1: @shadow nodes create two files, a **public** file without sentinels and a **private** file with sentinels.

Note 2: @file and @thin nodes are synonyms.

Note 3: Outline structure created in @edit nodes is not saved in the external file.

Within @path and @<file> paths, {{exp}} gets evaluated with the following symbols known: c, g, p, os and sys. For example:

```
@file {{os.path.abspath(os.curdir)}}/abc.py
```

refers to the file abc.py in (absolute path of) the current directory.

24.1.1 @asis <path>

The @asis directive creates an external file without sentinels and without any expansions.

Use this directive only when you must have complete control over every character of the external file. When writing @asis nodes, writes the body text of all nodes in outline order. Leo writes the body text *as is*, without recognizing section definitions, without expanding section references, and without treating directives specially in any way. In particular, Leo copies all directives, including @ or @c directives, to the external file as text.

The @@ convention: Within @asis trees only, if a headline starts with @@, Leo writes everything in the headline following the @@ just before the corresponding body text.

Files created from @asis trees contain *nothing* not contained in body text (or @@ headlines). In particular, if body text does not end in a newline, the first line from the next node will concatenated to the last line of the preceding node.

Within @asis trees, Leo writes no sentinels to the external file, so Leo can not update the outline using changes to the external file. When reading .leo files, Leo does *not* read external files created from @asis nodes. Instead, all data in an @asis tree is stored in the .leo file.

Within @asis trees, Leo recognizes the @ignore directive only in the *ancestors* of @asis nodes. This allows you to use the @ignore directive to prevent Leo from writing @asis trees.

Note: @file-asis and @silent are deprecated synonyms for @asis.

24.1.2 @auto <path>

The @auto directive imports an external file into a tree of nodes. Using @auto is *highly recommended* when using external files that must not contain Leo sentinels.

@auto trees allow people to use Leo in collaborative environments without using sentinels in external files. Even without sentinels, @auto trees can change when the corresponding external file changes outside of Leo.

When reading @auto nodes, Leo creates the @auto tree using **importers**, parsers that create an outline with nodes for each class, method and function in the external file. Some importers create other kinds of nodes as well.

Importers presently exist for C, elisp, HTML, .ini files, Java, Javascript, Pascal, PHP, Python and xml. Leo determines the language using the file's extension. If no parser exists for a language, Leo copies the entire body of the external file into the @auto node.

Note: the @data import_xml_tags setting specifies the **organizer tags** that cause the HTML and XML importers to create outline nodes. By default, the **organizer tags** are html, body, head, and div.

When writing @auto nodes, Leo writes the external file without sentinels. This allows you to use Leo in collaborative environments without disturbing colleagues.

When importing files into @auto trees, Leo performs several checks to ensure that writing the imported file will produce exactly the same file. These checks can produces **errors** or **warnings**. Errors indicate a potentially serious problem. Leo inserts an @ignore directive in the @auto tree if any error is found. This prevents the @auto tree from modifying the external file.

Before importing a file, Leo **regularizes** the leading whitespace of all lines of the original source file. That is, Leo converts blanks to tabs or tabs to blanks depending on the value of the @tabwidth directive in effect for the @auto node. Leo also checks that the indentation of any non-blank line is a multiple of the indentation specified by the @tabwidth directive. **Strict languages** are languages such as Python for which leading whitespace must be preserved exactly as it appears in the original source file. Problems during regularizing whitespace generate errors for strict languages and warnings for non-strict languages.

After importing a file, Leo verifies that writing the @auto node would create the same file as the original file. Such file comparison mismatches generate errors unless the problem involves only leading whitespace for non-strict languages. Whenever a mismatch occurs the first non-matching line is printed.

File comparison mismatches can arise for several reasons:

1. Bugs in the import parsers. Please report any such bugs immediately.
2. Underindented lines in classes, methods or function.

An **underindented line** is a line of body text that is indented less then the starting line of the class, method or function in which it appears. Leo outlines can not represent such lines exactly: every line in an external file will have at least the indentation of any unindented line of the corresponding node in the outline. Leo will issue a warning (not an error) for underindented Python comment lines. Such lines can not change the meaning of Python programs.

24.1.3 @edit <path>

The @edit directive imports an external file into a single node.

When reading @edit nodes, Leo reads the entire file into the @edit node. Lines that look like sentinels will be read just as they are.

When writing @edit nodes, Leo writes expands section references, @all and @others just as with @file trees. However, Leo writes no sentinels, so the structure created by sections references, @all and @others is not preserved.

24.1.4 @file <path> (aka @thin)

The @file directive creates an external file containing sentinels. When writing @file trees, Leo expands section references and @all and @others directives.

When reading external files created by @file, the sentinels allow Leo to recreate all aspects of the outline. In particular, Leo can update the outline based on changes made to the file by another editor.

Important: @file is the recommended way to create and edit most files. In particular, using @file nodes is **highly recommended** when sharing external files in a collaborative environment.

The @thin directive is a synonym for @file.

Prior to Leo 4.7, @file worked differently from @thin. This should not be a problem: Leo 4.7 can read all external files written by Leo 4.6.

24.1.5 @nosent <path>

The @nosent <filename> creates an external file without sentinel lines.

When writing an @nosent tree, Leo expands section references, @all and @others directives, but Leo writes no sentinels to the external file. Thus, Leo can not update @nosent trees from changes made to the external file. However, @nosent trees do have their uses: unlike @auto trees, cloned nodes *are* valid in @nosent trees.

When reading an @nosent node, Leo does *not* read the external file. Instead, all the data in the @nosent tree is stored in the .leo file.

Note: @auto or @shadow are usually better choices than @nosent for creating external files without sentinels.

Note: The @bool force_newlines_in_at_nosent_bodies setting controls whether Leo writes a trailing newline if non-empty body text does not end in a newline. The default is True.

24.1.6 @shadow <path>

The @shadow directive creates *two* external files, a **public** file without sentinels, and a **private** file containing sentinels.

When reading an @shadow node, Leo uses a brilliant algorithm devised by Bernhard Mulder that compares the public and private files, and then updates the outline based on changes to the *public* file. In this way, @shadow provides many of the benefits of @file trees without writing sentinels in the (public) external file.

Leo can do an initial import of @shadow trees by parsing the corresponding public file, exactly as is done for @auto nodes.

24.2 Part 2: @all and @others

These control how Leo places text when writing external files. They are two of the most important directives in Leo.

@all

Copies *all* descendant nodes to the external file. Use @all to place unrelated data in an external file.

The @all directive is valid only in the body of @file trees.

Within the range of an @all directive, Leo ignores the @others directive and section references, so Leo will not complain about orphan nodes.

@others

Writes the body text of all unnamed descendant into the external file, in outline order.

Whitespace appearing before @others directive adds to the indentation of all nodes added by the @others directive.

A single node may contain only one @others directive, but descendant nodes may have other @others directives.

24.3 Part 3: Syntax coloring directives

The @color, @killcolor, @nocolor and @nocolor-node directives control how Leo colors text in the body pane.

These directives typically affect the node in which they appear and all descendant nodes. Exception: an **ambiguous node**, a node containing both @color and @nocolor directives, has no effect on how Leo colors text in descendant nodes.

@color

Enables syntax coloring until the next @nocolor directive.

@killcolor

Disables syntax coloring in a node, overriding all @color, @nocolor or @nocolor-node directives in the same node.

@nocolor

Disables syntax coloring until the next @nocolor directive.

@nocolor-node

Disables coloring for only the node containing it. The @nocolor-node directive overrides the @color and @nocolor directives within the same node.

24.4 Part 4: Dangerous directives

These directives alter how Leo represents data in external files. They are **dangerous**--mistakes in using these sentinels can make it impossible for Leo to read the resulting external file. Use them with care!

Nevertheless, these sentinels can be useful in special situations.

@comment <1, 2 or three comment delims>

Sets the comment delimiters in @file and @shadow files. **Important**: Use @comment for unusual situations only. In most cases, you should use the @language directive to set comment delimiters.

The @comment directive may be followed by one, two or three delimiters, separated by whitespace. If one delimiter is given, it sets the delimiter used by single-line comments. If two delimiters are given, they set the block comment delimiter. If three delimiters are given, the first sets the single-line-comment delimiter, and the others set the block-comment delimiters.

Within these delimiters, underscores represent a significant space, and double underscores represent a newline. Examples:

```
@comment REM_
@comment __=pod__ __=cut__
```

The second line sets PerlPod comment delimiters.

Warning: the @comment and @delims directives **must not** appear in the same node. Doing so may create a file that Leo can not read.

Note: @language and @comment may appear in the same node, provided that @comment appears *after* the @language directive: @comment overrides @language.

The @comment directive must precede the first section name or @c directive.

@delims <1 or 2 comment delims>

Sets comment delimiters in external files containing sentinel lines.

The @delims directive requires one or two delimiters, separated by whitespace. If one delimiter is present it sets the single-line-comment delimiter. If two delimiters are present they set block comment delimiters.

This directive is often used to place Javascript text inside XML or HTML files. Like this:

```
@delims /* */
Javascript stuff
@delims <-- -->
HTML stuff
```

Warning: you **must** change back to previous delimiters using another @delims directive. Failure to change back to the previous delimiters will thoroughly corrupt the external file as far as compilers, HTML renderers, etc. are concerned. Leo does not do this automatically at the end of a node.

Warning: the @comment and @delims directives **must not** appear in the same node. Doing so may create a file that Leo can not read.

Note: The @delims directive can not be used to change the comment strings at the start of the external file, that is, the comment strings for the @+leo sentinel and the initial @+body and @+node sentinels.

@raw and @end_raw

@raw starts a section of ``raw'' text that ends *only* with the @end_raw directive or the end of the body text containing the @raw directive. Within this range, Leo ignores all section references and directives, and Leo generates no additional leading whitespace.

24.5 Part 5: All other directives

This section is a reference guide for all other Leo directives, organized alphabetically.

Unless otherwise noted, all directives listed are valid only in body text, and they must start at the leftmost column of the node.

@ and @doc

These directives start a doc part. @doc is a synonym for @. Doc parts continue until an @c directive or the end of the body text. For example:

```
@ This is a comment in a doc part.
Doc parts can span multiple lines.
The next line ends the doc part
@c
```

When writing external files, Leo writes doc parts as comments.

Leo does not recognize @ or @doc in @asis trees or when the @all or @delims directives are in effect.

@c and @code

Ends any doc part and starts a code part.

@code is a deprecated synonym for @c.

Leo does not recognize this directive in @asis trees or when the @all or @raw directives are in effect.

@chapter and @chapters

An @chapter tree represents a chapter. All @chapter nodes should be contained in an @chapters node.

These directives are too complex to describe here. For full details, see Using Chapters.

These directives must appear in the node's headline.

@encoding <encoding>

Specifies the Unicode encoding for an external file. For example:

```
@encoding iso-8859-1
```

When reading external files, the encoding given must match the encoding actually used in the external file or ``byte hash'' will result.

@first <text>

Places lines at the very start of an external file, before any Leo sentinels. @first lines must be the very first lines in an @<file> node. More then one @first lines may appear.

This creates two first lines, a shebang line and a Python encoding line:

```
@first #! /usr/bin/env python
@first # -*- coding: utf-8 -*-
```

Here is a perl example:

```
@first #!/bin/sh -- # perl, to stop looping
@first eval 'exec /usr/bin/perl -w -S $0 ${1+"$@"}'
@first     if 0;
```

@ignore

Tells Leo to ignore the subtree in which it appears.

In the body text of most top-level @<file> nodes, the @ignore directive causes Leo not to write the tree. However, Leo ignores @ignore directives in @asis trees.

Plugins and other parts of Leo sometimes @ignore for their own purposes. For example, Leo's unit testing commands will ignore trees containing @ignore. In such cases, the @ignore directive may appear in the headline or body text.

@language <language name>

Specifies the language in effect, including comment delimiters. If no @language directive is in effect, Leo uses the defaults specified by the @string target-language setting.

A node may contain at most one @language directive.

The valid language names are: actionscript, ada, autohotkey, batch, c, config, cpp, csharp, css, cweb, elisp, forth, fortran, fortran90, haskell, haxe, html, ini, java, javascript, kshell, latex, lua, noweb, pascal, perl, perlpod, php, plain, plsql, python, rapidq, rebol, rest, rst, ruby, shell, tcltk, tex, unknown, unknown_language, vim, vimoutline, xml, xslt.

Note: Shell files have comments that start with #.

Case is ignored in the language names. For example, the following are equivalent:

```
@language html
@language HTML
```

The @language directive also controls syntax coloring. For language x, the file leo/modes/x.py describes how to colorize the language. To see the languages presently supported, look in the leo/modes directory. There are over 100 such languages.

@last <text>

Places lines at the very end of external files.

This directive must occur at the very end of top-level @<file> nodes. More than one @last directive may exist. For example:

```
@first <?php
...
@last ?>
```

Leo does not recognize @last directive in @asis trees.

@lineending cr/lf/nl/crlf

Sets the line endings for external files. This directive overrides the @string output_newline setting.

The valid forms of the @lineending directive are:

@lineending nl	The default, Linux.
@lineending cr	Mac
@lineending crlf	Windows
@lineending lf	Same as `nl', not recommended
@lineending platform	Same as platform value for output_newline setting.

@nowrap

Disables line wrapping the Leo's body pane.

Only the first @wrap or @nowrap directive in a node has any effect.

@nowrap may appear in either headlines or body text.

@pagewidth <n>

Sets the page width used to break doc parts into lines. <n> should be a positive integer. For example:

```
@pagewidth 100
```

The @pagewidth directive overrides the @int page_width setting.

@path <path>

Sets the **path prefix** for relative filenames for all @<file> tree.

This directive may appear in headlines or body text, and may appear in top-level @<file> nodes.

The path is an **absolute path** if it begins with c:\ or /, otherwise the path is a **relative** paths.

Multiple @path directives may contribute to the path prefix. Absolute paths overrides any ancestor @path directives. Relative paths add to the path prefix.

If no @path directives are in effect, the default path prefix is the directory containing the .leo file.

Within @path and @<file> paths, {{exp}} gets evaluated with the following symbols known: c, g, p, os and sys. For example:

```
@file {{os.path.abspath(os.curdir)}}/abc.py
```

refers to the file abc.py in (absolute path of) the current directory.

@tabwidth <n>

Sets the width of tabs. Negative tab widths cause Leo to convert tabs to spaces.

@wrap

Enables line wrapping in Leo's body pane.

Only the first @wrap or @nowrap directive in a node has any effect.

@wrap may appear in either headlines or body text.

第 **25** 章

Designing with Leo

This chapter discusses how outlines can improve the design of programs, web sites and any other complex data. The more complex your program or data, the more useful Leo becomes.

Furthermore, the same features that help with design also help with implementation, maintenance and testing. In all cases, being able to organize, understand and manipulate (script) complex data is what Leo does best.

Contents

- Designing with Leo
  - Outlines embody design
  - Nodes hide details
  - Clones create views
  - Nodes create context

25.1 Outlines embody design

Leo's outlines don't merely represent design. They often *are* the design. Outlines effortlessly show relationships between class, methods or any other data.

Outlines express design and structure directly. In most programs, the grouping of functions into files, or the organization of a single file as a set of functions, etc.

Typical design tools are separate from the resulting product. With Leo, your designs become your programs or web site.

Leo's outline pane always shows you the big picture, and the relationship of the presently selected outline to that big picture. At all times you are aware of both the overall design and all the intermediate levels of detail.

Outlines create new design dimensions.

There are many ways to express a program as a Leo outline. Such choices are important. They add clarity to the entire program. These are different *kind* of choices. They simply can not be expressed at all in other editors. In other words, such choices exist in a new design space.

Outlines add a new dimension to the design and coding process. Choices about what sections do, what they are named, what order they appear in, are choices in a design space different from ``normal" programming. This an abstract concept, to be sure. However, the previous paragraphs are really a manifestation of working in this new design space.

25.2 Nodes hide details

Organizer nodes convey information about the structure and design of a large system. Decoupling structure from content in this way is precisely what is needed for flexibility: one can reorganize at will without worrying about changing the meaning of the code.

Outlines clarify the shape of code

These last several paragraphs have discussed comments in detail because the net effect of ``putting comments where they belong" is that comments don't clutter the code. Section references hide irrelevant detail, so larger-scale patterns within functions (or declarations) become more apparent. Often just recasting code into web format has created Aha's about my own code, with no special attention to recoding or redesign! Recasting a function as a web raises the real and apparent level of abstraction.

Organizer nodes eliminate mundane comments The headline of an organizer node is often all that needs to be said.

Nodes create places for comments

Each of Leo's nodes provide a place for lengthy comments that do not clutter other code. In practice this encourages comments where they are needed.

Outlines reduce the need for comments

Bridge or transition phrases are almost always unnecessary in a Leo outline. One never needs to say something like, ``having just finished with topic x, we turn now to topic y."

Comments and formatting no longer have to indicate overall design; node do that.

Nodes reduce the visual ``weight" of code

Nodes and their helpers (usually in child nodes) can be as complex as necessary without affecting the organization of the outline. Moreover, collapsed nodes are inconspicuous, no matter how many children they contain and no matter how much code each child contains.

25.3 Clones create views

Typical browsers show you a fixed view of code. In contrast, Leo allows you to organize your programs as *you* want, and *Leo remembers your organizations*.

Furthermore, Leo does not constrain you to a single ``right" view of your data, programs or designs. By using clones, a single outline may contain dozens or even thousands of views of the nodes in the outline. You are free to create a new view for every new task or project.

Finally, you can insert organizer nodes anywhere in an outline, including anywhere in any view. Organizer nodes do not change the meaning of programs, html pages or web sites, yet they can clarify and simplify designs, programs, web sites or data.

25.4 Nodes create context

Nodes naturally provide **useful** context. For example, @button and @test nodes.

Outlines provide a convenient way of expressing the intended scope of commands. Many of Leo's commands operates on the presently selected tree.

第 **26** 章

History of Leo

This chapter discusses the history of Leo and tells the essential features of each version. Here are the most important dates in Leo's history:

Contents

- History of Leo
  - Beginnings
  - Breakthrough
  - Apple and YellowBox
  - Borland C++
  - Discovering Python
  - SourceForge
  - Allowing sentinel lines in external files
  - Untangling @file is easy!
  - Leo 3.x: Continuous improvement
  - Leo 4.0: Eliminating error `recovery'
  - Leo 4.1: The debut of gnx's
  - Leo 4.2: Complete at last
  - Leo 4.3 Settings
  - Leo 4.4 The minibuffer and key bindings
  - Leo 4.4.x Improvements
  - Leo 4.5 @shadow files
  - Leo 4.6 Caching, Qt and more
  - Leo 4.7 The one node world and Python 3k
  - Leo 4.8 Simple sentinels & better data recovery

26.1 Beginnings

Leo grew out of my efforts to use Donald Knuth's ``CWEB system of Structured documentation." I had known of literate programming since the mid 1980's, but I never understood how to make it work for me. In November 1995 I started thinking about programming in earnest. Over the holidays I mused about making programs more understandable. In January 1996 the fog of confusion suddenly cleared. I summarized my thinking with the phrase, **web are outlines in disguise**. I suspected that outline views were the key to programming, but many details remained obscure.

26.2 Breakthrough

March 5, 1996, is the most important date in Leo's history. While returning from a day of skiing, I discussed my thoughts with Rebecca. During that conversation I realized that I could use the MORE outliner as a prototype for a ``programming outliner." I immediately started work on my first outlined program. It quickly became apparent that outlines work: all my old problems with programming vanished. The @others directive dates from this day. I realized that MORE's outlines could form the basis for Leo's screen design. Rather than opening body text within the outline, as MORE does, I decided to use a separate body pane.

I hacked a translator called M2C which allowed me to use MORE to write real code. I would write code in MORE, copy the text to the clipboard in MORE format, then run M2C, which would convert the outline into C code. This process was useful, if clumsy. I called the language used in the outline SWEB, for simplified CWEB. Much later Leo started supporting the noweb language.

26.3 Apple and YellowBox

Throughout 1996 I created a version of Leo on the Macintosh in plain C and the native Mac Toolbox. This was a poor choice; I wasted a huge amount of time programming with these primitive tools. However, this effort convinced me that Leo was a great way to program.

Late in 1997 I wrote a Print command to typeset an outline. Printing (Weaving) is supposedly a key feature of literate programming. Imagine my surprise when I realized that such a ``beautiful" program listing was almost unintelligible; all the structure inherent in the outline was lost! I saw clearly that typesetting, no matter how well done, is no substitute for explicit structure.

In 1998 I created a version of Leo using Apple's YellowBox environment. Alas, Apple broke its promises to Apple developers. I had to start again.

26.4 Borland C++

I rewrote Leo for Borland C++ starting in May 1999. Borland C++ was much better than CodeWarrior C, but it was still C ++. This version of Leo was the first version to use xml as the format of .leo files. The last version of Borland Leo, 3.12 Final went out the door July 17, 2003.

26.5 Discovering Python

I attended the Python conference in early 2001. In May of 2000 I began work on an wxWindows version of Leo. This did not work out, but something good did come from this effort. I spent a lot of time adding Python scripting to the wxWindows code and I became familiar with Python and its internals.

I really started to `get' Python in September 2001. I wrote the white papers at about this time. Python solved *all* my programming problems. I rewrote Leo in Python in about two months! For the first time in my career I was no longer anxious while programming; it simply isn't possible to create bad bugs in Python. The Python version of Leo was the first officially OpenSoftware version of Leo. The first functional version of Leo in Python was 0.05 alpha, December 17, 2001.

26.6 SourceForge

I registered the Leo project on SourceForge on March 10, 2003. It is certainly no accident that Leo started a new life shortly thereafter. Prior to SourceForge my interest in Leo had been waning.

26.7 Allowing sentinel lines in external files

In the summer of 2001 I began to consider using sentinel lines in external files. Previously I had thought that outline structure must be `protected' by remaining inside .leo files. Accepting the possibility that sentinels might be corrupted opened vast new design possibilities. In retrospect, problems with sentinel almost never happen, but that wasn't obvious at the time! The result of this design was known at first as Leo2. That terminology is extinct. I think of this version as the first version to support @file and automatic tangling and untangling.

26.8 Untangling @file is easy!

The biggest surprise in Leo's history was the realization it is **much** easier to untangle files derived from @file. Indeed, the old tangle code created all sorts of problems that just disappear when using @file. The new Python version of Leo became fully operational in early 2002. It was probably about this time that I chose noweb as Leo's preferred markup language. My decision not to support noweb's escape sequences made Leo's read code much more robust.

26.9 Leo 3.x: Continuous improvement

I spent 2002 taking advantages of Python's tremendous power and safety. Many improvements were at last easy enough to do:

- Nested @others directives appeared in 3.2.
- Unicode support started in 3.3.
- @first and @last appeared in 3.7
- @asis and @nosent appeared in 3.8.
- Incremental syntax coloring and incremental undo appeared in 3.9.

- Paul Paterson created Leo's plugin architecture sometime during this period. Plugins have been a driving force in Leo's development because people can change how Leo works without altering Leo's core.
- 3.12 fixed a huge memory leak.
- 3.12 Final, the last 3.x version, appeared July 17, 2003.

26.10 Leo 4.0: Eliminating error `recovery'

In late 2002 and throughout 2003 I worked on an entirely new file format. 4.0 final went out the door October 17, 2003 after almost a year intense design work trying to improve error recovery scheme used while reading external files. In the summer of 2003 I realized that orphan and @ignore'd nodes must be prohibited in @file trees. With this restriction, Leo could finally recreate @file trees in outlines using **only** the information in external files. This made the read code much more robust, and eliminated all the previous unworkable error recovery schemes. At last Leo was on a completely firm foundation.

26.11 Leo 4.1: The debut of gnx's

Leo first used gnx's (global node indices) as a foolproof way of associating nodes in .leo files with nodes in external files. At the time, there was still intense discussions about protecting the logical consistency of outlines. @thin was later to solve all those problems, but nobody knew that then.

26.12 Leo 4.2: Complete at last

Leo 4.2 Final went out the door September 20, 2004. This surely is one of the most significant dates in Leo's history:

- This marked the end worries about consistency of outlines and external files: Leo recreates all essential information from thin external files, so *there is nothing left in the .leo file to get out of synch*.
- **Thin external files use gnx's extensively. This simplifies the file format and** makes thin external files more cvs friendly.
- A sensational scripting plugin showed how to create script buttons. This has lead to improvements in the Execute Script command and other significant improvements in Unit testing.
- As if this were not enough, 4.2 marked the `great divide' in Leo's internal data structures. Before 4.2, Leo every node in the outline had its own vnode. This was a big performance problem: clone operations had to traverse the entire outline! 4.2 represents clones by sharing subtrees. Changing Leo's fundamental data structures while retaining compatibility with old scripts was engineering work of which the entire Leo community can be proud. Scripting Leo with Python tells how the position class makes this happen. This was a cooperative effort. Kent Tenney and Bernhard Mulder made absolutely crucial contributions. Kent pointed out that it is a tnode, not a vnode that must form the root of the shared data. Bernhard showed that iterators are the way to avoid creating huge numbers of positions.

Leo 4.2 marked so many significant changes. I often find it hard to remember what life with Leo was like before it.

26.13 Leo 4.3 Settings

Leo 4.3 corrected many problems with leoConfig.txt. Instead, Leo gets settings from one or more leoSettings.leo files. This version also introduced a way to changed settings using a settings dialog. However, the settings dialog proved not to be useful (worse, it inhibited design) and the settings dialog was retired in Leo 4.4.

26.14 Leo 4.4 The minibuffer and key bindings

Leo 4.4 was a year-long effort to incorporate an Emacs-style minibuffer and related commands into Leo. Thinking in terms of minibuffer commands frees my thinking. Leo 4.4 also featured many improvements in how keys are bound to commands, including per-pane bindings and user-defined key-binding modes.

Development on long-delayed projects accelerated after 4.4 final went out the door. Recent projects include:

- Controlling syntax coloring with jEdit's xml language-description files.
- Support for debugging scripts using external debuggers.
- Modifying Leo's vnodes and tnodes so that Leo's data can be used with ZODB.
- Using pymacs to write Leo scripts within Emacs.
- Using the leoBridge module to embed Leo support in other programs.
- Using Leo to run unit tests.

26.15 Leo 4.4.x Improvements

This series of releases featured **hundreds** of improvements. The highlights were truly significant:

- Added the leoBridge module. See Embedding Leo with the leoBridge Module.
- Added support for @enabled-plugins and @open-with nodes in settings files.
- Added support for ZODB. See Using ZODB with Leo.
- Added leoPymacs module. See Leo and Emacs.
- Added perfect import of external files with @auto nodes.
- Used the sax parser to .leo files. This allows the format of .leo files to be expanded easily.
- Added support for myLeoSettings.leo.
- Supported multiple editors in body pane.
- Added the jEdit_colorizer plugin. See Controlling Syntax Coloring.
- Many other new plugins.

For a complete list, see the What's New chapter.

26.16 Leo 4.5 @shadow files

Added support for @shadow files. This was a major breakthrough. See the Using @shadow chapter for full details.

26.17 Leo 4.6 Caching, Qt and more

This version of Leo featured more significant improvements:

- Added support for the Qt gui. This was a major project that significantly improves the look and feel of Leo.
- A file-caching scheme produced spectacular improvements in the speed of loading Leo outlines.
- Added support for @auto-rst nodes. These import reStructuredText (rST) files so that the files can be ``round-tripped'' without introducing extraneous changes. This makes Leo a superb environment for using rST.
- Added support for @edit nodes.

26.18 Leo 4.7 The one node world and Python 3k

Leo 4.7 accomplishes something I long thought to be impossible: the unification of vnodes and tnodes. tnodes no longer exist: vnodes contain all data. The Aha that made this possible is that iterators and positions allow a single node to appear in more than one place in a tree traversal.

This is one of the most significant developments in Leo's history. At last the endless confusion between vnodes and tnodes is gone. At the most fundamental level, Leo's data structures are as simple as possible. This makes them as general and as powerful as possible!

This version successfully produced a common code base that can run on both Python 2.x and Python 3.x.

26.19 Leo 4.8 Simple sentinels & better data recovery

Leo 4.8 simplified Leo's sentinels as much as possible. Leo's sentinel lines look very much like Emacs org-mode comment lines, except for the addition of gnx's.

This version also produced a fundamentally important addition to Leo's error recovery. Leo now shows ``Resurrected'' and ``Recovered'' nodes when loading an outline. These nodes protect against data loss, and also implicitly warn when unusual data-changing events occur. Creating this scheme is likely the final chapter in the epic saga of error recovery in Leo.

第 27 章

Theory of Operation

This chapter discusses how Leo's code works, paying particular attention to topics that have caused difficulties in design or implementation. This chapter will be of use primarily to those wanting to change Leo's code.

Contents

- Theory of Operation
  - Autocompletion
  - Clones
  - Drawing and events
  - Find and change commands
  - Key handling
  - Nodes
  - Overview
  - Unicode
  - Unlimited undo
  - Leo's load process

27.1 Autocompletion

27.1.1 UI notes

Both the legacy and new completer now work *exactly* the same way, because they both use the AutoCompleterClass to compute the list of completions.

The strict ``stateless'' requirement means that the ``intermediate'' completions must be entered into the body pane while completion is active. It works well as a visual cue when using the tabbed completer: indeed, the tabbed completer would be difficult to use without this cue.

The situation is slightly different with the qcompleter. Adding code before the user accepts the completion might be considered an ``advanced" feature. However, it does have two important advantages, especially when ``chaining" across periods: it indicates the status of the chaining and it limits what must appear in the qcompleter window.

27.1.2 Appearance

There is little change to the legacy completer, except that no text is highlighted in the body pane during completion. This is calmer than before. Furthermore, there is no longer any need for highlighting, because when the user types a backspace the legacy completer now simply deletes a single character instead of the highlighted text.

One minor change: the legacy completer now *does* insert characters that do not match the start of any possible completion. This is an experimental feature, but it might play well with using codewise completions as a fallback to Leo-related completions.

27.1.3 Performance

Performance of Leo-related completions is *much* better than before. The old code used Python's inspect module and was horribly complex. The new code uses eval and is perfectly straightforward.

The present codewise-related code caches completions for all previously-seen prefixes. This dramatically speeds up backspacing. Global caching is possible because completions depend *only* on the present prefix, *not* on the presently selected node. If ContextSniffer were used, completions would depend on the selected node and caching would likely be impractical. Despite these improvements, the performance of codewise-oriented completions is noticeably slower than Leo-related completions.

The ac.get_cached_options cuts back the prefix until it finds a cached prefix. ac.compute_completion_list then uses this (perhaps-way-too-long-list) as a starting point, and computes the final completion list by calling g.itemsMatchingPrefixInList.

This may not be absolutely the fastest way, but it is much simpler and more robust than attempting to do ``prefix AI" based on comparing old and new prefixes. Furthermore, this scheme is completely independent of the how completions are actually computed. The autocompleter now caches options lists, regardless of whether using eval or codewise.

In most cases the scheme is extremely fast: calls to get_completions replace calls to g.itemsMatchingPrefixInList. However, for short prefixes, the list that g.g.itemsMatchingPrefixInList scans can have thousands of items. Scanning large lists can't be helped in any case for short prefixes.

Happily, the new scheme is still *completely* stateless: the completionDict does *not* define state (it is valid everywhere) and no state variables had to be added. In short, the new caching scheme is much better than before, and it probably is close to optimal in most situations.

27.2 Clones

New in Leo 4.7. All clones of a node **are the same node**. This is the so-called **one-node** world. In this world, vnodes represent data, generators and positions represent the location of the data in an outline. This is a much simpler world than all previous data representations.

In Leo versions 4.2 to 4.6 clones were represented by sharing **tnodes**. Cloned vnodes shared the same tnode. This shared tnode represented the entire shared subtree of both clones. Thus, the _firstChild link had to reside in *tnodes*, not *vnodes*.

Prior to Leo version 4.2, Leo duplicated all the descendants of vnode v when cloning v. This created many complications that were removed in the shared tnode world. In particular, in the shared tnode scheme a vnode v is cloned if and only if len (v.vnodeList) > 1.

27.3 Drawing and events

Leo must redraw the outline pane when commands are executed and as the result of mouse and keyboard events. The main challenges are eliminating flicker and handling events properly. These topics are interrelated.

Eliminating flicker. Leo must update the outline pane with minimum flicker. Various versions of Leo have approached this problem in different ways. The drawing code in leo.py is robust, flexible, relatively simple and should work in almost any conceivable environment. Leo assumes that all code that changes the outline pane will be enclosed in matching calls to the c.beginUpdate and c.endUpdate methods of the Commands class. c.beginUpdate() inhibits drawing until the matching c.endUpdate(). These calls may be nested; only the outermost call to c.endUpdate() calls c.redraw() to force a redraw of the outline pane.

Code may call c.endUpdate(flag) instead of c.endUpdate(). Leo redraws the screen only if flag is true. This allows code to suppress redrawing entirely when needed. For example, here is how the idle_body_key event handler in leoTree.py conditionally redraws the outline pane:

```
redraw_flag = false
c.beginUpdate()
val = v.computeIcon()
if val != v.iconVal:
    v.iconVal = val
    redraw_flag = true
c.endUpdate(redraw_flag) # redraw only if necessary
```

The leoTree class redraws all icons automatically when c.redraw() is called. This is a major simplification compared to previous versions of Leo. The entire machinery of drawing icons in the vnode class has been eliminated. The v.computeIcon method tells what the icon should be. The v.iconVal ivar tells what the present icon is. The event handler simply compares these two values and sets redraw_flag if they don't match.

Handling events. Besides redrawing the screen, Leo must handle events or commands that change the text in the outline or body panes.

The leoTree class contains all the event handlers for the body and outline panes. The actual work is done in the idle_head_key and idle_body_key methods. These routines are surprisingly complex; they must handle all the tasks mentioned above, as well as others. The idle_head_key and idle_body_key methods should not be called outside the leoTree class. However, it often happens that code that handles user commands must simulate an event. That is, the code needs to indicate that headline or body text has changed so that the screen may be redrawn properly. The leoTree class defines the following simplified event handlers: onBodyChanged, onBodyWillChange, onBodyKey, onHeadChanged and onHeadlineKey. Commanders and subcommanders call these event handlers to indicate that a command has changed, or will change, the headline or body text. Calling event handlers rather than c.beginUpdate and c.endUpdate ensures that the outline pane is redrawn only when needed.

27.4 Find and change commands

The find and change commands are tricky; there are many details that must be handled properly. The following principles govern the LeoFind class:

1. Find and Change commands initialize themselves using only the state of the present Leo window. In particular, the Find class must not save internal state information from one invocation to the next. This means that when the user changes the nodes, or selects new text in headline or body text, those changes will affect the next invocation of any Find or Change command. Failure to follow this principle caused all kinds of problems in the Borland and Macintosh codes. There is one exception to this rule: we must remember where interactive wrapped searches start. This principle simplifies the code because most ivars do not persist. However, each command must ensure that the Leo window is left in a state suitable for restarting the incremental (interactive) Find and Change commands. Details of initialization are discussed below.
2. The Find and Change commands must not change the state of the outline or body pane during execution. That would cause severe flashing and slow down the commands a great deal. In particular, the c.selectPosition and c.editPosition methods must not be called while looking for matches.
3. When incremental Find or Change commands succeed they must leave the Leo window in the proper state to execute another incremental command. We restore the Leo window as it was on entry whenever an incremental search fails and after any Find All and Change All command. Initialization involves setting the self.c, self.v, self.in_headline, self.wrapping and self.s_text ivars.

Setting self.in_headline is tricky; we must be sure to retain the state of the outline pane until initialization is complete. Initializing the Find All and Change All commands is much easier because such initialization does not depend on the state of the Leo window. Using the same kind of text widget for both headlines and body text results in a huge simplification of the code.

Indeed, the searching code does not know whether it is searching headline or body text. The search code knows only that self.s_text is a text widget that contains the text to be searched or changed and the insert and sel attributes of self.search_text indicate the range of text to be searched. Searching headline and body text simultaneously is complicated. The selectNextVnode() method handles the many details involved by setting self.s_text and its insert and sel attributes.

27.5 Key handling

The following three sections deal with different aspects of how Leo handle's keystrokes that the user types. This is the most complex code in Leo.

27.5.1 Key domains

Leo's key-handling code has almost nothing to do with supporting multiple guis. Rather, the key-handling code is complex because it must deal with the following four fundamentally different problem domains.

Domain 1: Parsing user bindings in @keys, @mode and @shortcut nodes

In this domain, complexity arises from allowing the user a variety of *equivalent* ways of specifying bindings. Furthermore, this domain allows the user to specify modes and per-pane bindings. Thus, all these complexities are unavoidable.

Domain 2: Maintaining and using binding tables

The result of parsing user bindings are a set of binding tables. These tables are complex, but we need not go into details here because only one method, k.masterKeyHandler (and its helper, getPaneBinding) uses the tables.

The only thing we have to remember about the binding tables is that bindings are expressed in terms of so-called **strokes**. Strokes are the ``official'' form of every user binding. The essential property of a stroke is that it contains *all* the information required to handle the stroke: Leo can unambiguously determine exactly what a stroke means and what bindings are in effect for a stroke. If necessary, Leo can correctly insert the proper character *corresponding* to the stroke into any text widget.

This correspondence (association) between the stroke and the actual character to be inserted into text widgets is crucial. This correspondence is created in the next domain.

Anticipating a bit, for any incoming key event, event.stroke is the stroke, and event.char is the character (if any) that *might* (depending on bindings) be inserted into text widgets.

New in Leo 4.9: k.stroke2char calculates the to-be-inserted char from any stroke.

Domain 3: Translating incoming key events into standard events

The eventFilter method in qtGui.py creates leoKeyEvent objects. Turning ``raw'' Qt key events into leoKeyEvents is unavoidably complicated because eventFilter (and its allies) **must** carefully compute the stroke corresponding to the raw key event. There is no way around this requirement if Leo's binding machinery in domain 2 is to work. This code has been stable for a long time.

Domain 4: Printing key bindings in a human-readable format

It's important not to forget this domain: there are some situations in which we want to represent `b','r','n' and `t' as `BackSpace','Linefeed','Return' and `Tab'.

27.5.2 Key bindings

There are two kinds of bindings, gui bindings and pane bindings.

Gui bindings are the actual binding as seen by whatever gui is in effect. Leo binds every key that has a binding to k.masterKeyHander.

At present Leo makes gui bindings in several places, all equivalent. Bindings are made to callbacks, all of which have this form:

```
def callback(event=None,k=k,stroke=stroke):
  return k.masterKeyHandler(event,stroke)
```

As a result, changing gui bindings actually has no effect whatever. It would be clearer to have a single place to make these bindings...

In any case, the purpose of these callbacks is to capture the value of `stroke' so that it can be passed to k.masterKeyHandler. This relieves k.masterKeyHandler of the impossible task of computing the stroke from the event.

Important: No function argument is ever passed to k.masterKeyHandler from these callbacks, because k.masterKeyHandler binds keys to command handlers as described next.

Pane bindings are bindings represented by various Python dictionaries in the keyHandlerClass (see below). k.masterKeyHandler and its helpers use these dictionaries to call the proper command or mode handler. This logic is hairy, but it is completely separate from the gui binding logic.

Here are the dictionaries that k.masterKeyHandler uses:

- c.commandsDict: Keys are minibuffer command names; values are functions f.
- k.inverseCommandsDict: Keys are f.__name__l values are emacs command names.
- k.bindingsDict: Keys are shortcuts; values are *lists* of g.bunch(func,name,warningGiven).
- k.masterBindingsDict: Keys are pane names: `all','text',etc. or mode names. Values are dicts: keys are strokes; values are g.Bunch(commandName,func,pane,stroke).
- k.modeWidgetsDict: Keys are mode names; values are lists of widgets to which bindings have been made.
- k.settingsNameDict: Keys are lowercase settings; values are `real' Tk key specifiers. Important: this table has no inverse.
- inverseBindingDict: This is *not* an ivar; it is computed by k.computeInverseBindingDict(). Keys are emacs command names; values are *lists* of shortcuts.

27.5.3 Handling key events

All events are key events

All event objects passed around Leo are *key* event objects. Taking a look at the eventFilter method, we see clearly see that *only* key events ever get passed to Leo's core. All other events are handled by Qt-specific event handlers.

As can be seen, these non-key events *can* be passed to Leo, but only as the event arg in g.doHook (!) At present, no plugin ever calls k.masterKeyHandler. The only call to k.masterKeyHandler in qtGui.py is the expected call in eventFilter.

There are other calls to k.masterKeyHandler in Leo's core, but we can prove (by induction, if you will), that all events passed to k.masterKeyHandler are proper leoKeyEvent objects.

c.check_event

The essential invariant is that the events passed to Leo's core methods really are leoKeyEvent objects created by qtGui.py. Rather than asserting this invariant, the code will contains calls to c.check_event in essential places. c.check_event is a ``relaxed" place to do as much error checking is needed. In particular, running the unit tests calls c.check_event many times.

c.check_event is a happy ``accident". It turns out to be the essential consistency check that continually verifies that the Qt event methods are delivering the expected keys to k.masterKeyHandler.

27.5.4 About the KeyStroke class

The KeyStroke class distinguish between ``raw" user settings and the ``canonicalized" form used throughout Leo. Indeed, the ability to explicitly distinguish between the two, using type checking, substantially clarifies and simplifies the code.

The KeyStroke class makes possible vital type-related assertions. Knowing *for sure* exactly what crucial data is and what it means is a huge step forward.

Objects of the KeyStroke class can be used *exactly* as a strings may be used:

A. KeyStroke objects may be used as dictionary keys, because they have __hash__ methods and all the so-called rich comparison methods: __eq__, __ne__, __ge__, __gt__, __le__ and __lt__. Note that KeyStroke objects may be compared with other KeyStroke objects, strings and None.

B. At present, KeyStroke objects supports the find, lower and startswith methods. This simplifies the code substantially: we can apply these methods to either strings or KeyStroke objects, so there is no need to create different versions of the code depending on the value of g.new_strokes.

However, having the KeyStroke class support string methods is bad design. Indeed, it is a symptom that the client code that uses KeyStroke objects knows too much about the internals of KeyStroke objects. Instead, the KeyStroke class should have higher-level methods that use s.find, s.lower and s.startswith internally.

You could say that the fact that code in leoKeys.py calls s.find, s.lower and s.startswith is a symptom of non OO programming. The internal details of settings and strokes ``pollutes" the code. This must be fixed. This will likely create opportunities for further simplifications.

> Why not just have .s attribute in KeyStroke, that contains the string version?

It is truly impossible to understand the key code without knowing whether an object is a string representing a user setting or the canonicalized version used in Leo's core, that is, a KeyStroke object. Using ks.s instead of ks destroys precisely the information needed to understand the code.

Again, this is not a theoretical concern. The key code now contains assertions of the form:

```
assert g.isStroke(stroke)
```

or:

```
assert g.isStrokeOrNone(stroke)
```

Getting these assertions to pass in *all* situations required several important revisions of the code. The code that makes the assertions pass is ``innocuous", that is, almost invisible in the mass of code, but obviously, these small pieces of code are vital.

This is in no way a violation of OO principles. The code is not dispatching on the type of objects, it is merely enforcing vital consistency checks. This code is complex: confusion about the types of objects is intolerable. Happily, the resulting clarity allows the code to be substantially simpler than it would otherwise be, which in turn clarifies the code further, and so on...

27.5.5 Simplifying the Qt input code

The Qt key input code can be greatly simplified by calling a new k.makeKeyStrokeFromData factory method. At present, the Qt key input code knows *all* the details of the format of *canonicalized* settings. This is absolutely wretched design.

Instead, the Qt input key code should simply pass the key modifiers and other key information to k.makeKeyStrokeFromData, in a some kind of ``easy" format. For example, the Qt input key code would represent the internal Qt modifiers as lists of strings like ``alt", ``ctrl", ``meta", ``shift". k.makeKeyStrokeFromData would then create a *user* setting from the components, and then call k.strokeFromSetting to complete the transformation.

27.5.6 Typed dicts

Leo's key dictionaries will always be complex, but basing them on the TypedDict class is a major improvement.

The g.TypedDict and g.TypedDictOfLists classes are useful for more than type checking: they have unique names and a dump method that dumps the dict in an easy-to-read format that includes the name, and valid types for keys and values.

Plain dicts do have their uses, but for ``long-lived" dicts, and dicts passed around between methods, plain dicts are as ill-advised as g.Bunches.

27.6 Nodes

The vnode class is Leo's fundamental model class. A **vnode** represents the data represented by headlines. As of Leo 4.7, all clones of a node are in fact **exactly the same node**.

The vnode contains **all** data associated with a node. A vnode contains headline text, body text, and **user attributes**, uA's for short.

Because Leo has unlimited Undo commands, Leo deletes vnodes only when a window closes. Leo deletes nodes indirectly using destroy methods. Several classes, including the vnode, leoFrame and leoTree classes, have destroy methods. destroy methods merely clear links so that Python's reference counting mechanisms will eventually delete vnodes and other data when a window closes.

Leo's XML file format uses **tx** and **t** attributes to associate <v> elements with <t> elements. <v> elements represent nodes. <t> elements represent the body text of nodes. This is a (somewhat dubious) space optimization. The values of tx and t attributes are **gnx's** (global node indices). These indices do not change once a node has created.

27.7 Overview

All versions of Leo are organized as a collection of classes. The general organization of Leo has remained remarkably stable throughout all versions of Leo, although the names of classes are different in different versions. Smalltalk's Model/View/Controller terminology is a good way think about Leo's classes. **Model classes** represent the fundamental data. The vnode class is Leo's primary model class.

View classes draw the screen. The main view classes are leoFrame.py and leoTree.py. The colorizer class in leoColor.py handles syntax coloring in the body pane. Leo's view classes know about data stored in the vnode class. Most events (keystrokes and mouse actions) in the outline and body pane are handled in the leoTree class. The leoFrame class also creates the Leo window, including menus, and dispatches the appropriate members of the controller classes in response to menu commands.

Controller classes (aka commanders) control the application. In Leo, controllers mostly handle menu commands. Commanders create subcommanders to handle complex commands. The atFile class reads and writes files derived from @file trees. The LeoFind class handles the Find and Change commands. The leoImportCommands class handles the Import and Export commands, and the undoer class handles the Undo command. Other classes could be considered controller classes.

Each Leo window has its own commander and subcommanders. Subcommanders are not subclasses of their commander. Instead, subcommanders know the commander that created them, and call that commander as needed. Commanders and subcommanders call the model and view classes as needed. For example, the Commands class handles outline commands. To move a headline, the commander for the window calls a vnode move routine to alter the data, then calls the view class to redraw the screen based on the new data.

A singleton instance of the **LeoApp** class represents the application itself. All code uses the app() global function to gain access to this singleton member. The ivars of the LeoApp object are the equivalent of Leo's global variables. leo.py uses no global Python variables, except the gApp variable returned by app(). leoGlobals.py defines all application constants. Naturally, most constants are local to the class that uses them.

Several classes combine aspects of model, view and controller. For example, the **LeoPrefs** class represents user preferences (model), the Preference Panel (view) and the Preferences menu command (controller). Similarly, the **LeoFind** class represents find settings, the Find/Change dialog, and the Find/Change commands.

We use the following convention throughout this documentation. Any variable named c is a commander, i.e., an instance of the Commands class in leoCommands.py. Variables named v are vnodes. These classes are defined in leoNodes.py.

27.8 Unicode

Leo uses unicode objects in vnodes to denote headline and body text. Note that unicode strings have no encoding; only plain strings have encodings. This means that once an (encoded) plain string has been converted to a unicode string it doesn't matter how the unicode string was created. This is the key that makes Leo's new code robust: internally Leo never has to worry about encodings. Encoding matter only when encoded strings are converted to and from Unicode. This happens when Leo reads or writes files.

Python expressions that mix unicode strings u and plain strings s, like one of these:

```
u + s
u == s
u[5] == s[2:]
```

are promoted to unicode objects using the ``system encoding". This encoding should never be changed, but we can't assume that we know what it is, so for safety we should assume the most restrictive encoding, namely ``ascii". With this assumption, Leo's code can't throw an exception during these promotions provided that:

- Leo converts all text to unicode when Leo reads files or gets text from text widgets.
- All string literals in Leo's code have only ascii characters.

27.9 Unlimited undo

Unlimited undo is straightforward; it merely requires that all commands that affect the outline or body text must be undoable. In other words, everything that affects the outline or body text must be remembered. We may think of all the actions that may be Undone or Redone as a string of beads (undo nodes).

Undoing an operation moves backwards to the next bead; redoing an operation moves forwards to the next bead. A bead pointer points to the present bead. The bead pointer points in front of the first bead when Undo is disabled. The bead pointer points at the last bead when Redo is disabled. An undo node is a Python dictionary containing all information needed to undo or redo the operation. The Undo command uses the present bead to undo the action, then moves the bead pointer backwards.

The Redo command uses the bead after the present bead to redo the action, then moves the bead pointer forwards. All undoable operations call setUndoParams() to create a new bead. The list of beads does not branch; all undoable operations (except the Undo and Redo commands themselves) delete any beads following the newly created bead. I did not invent this model of unlimited undo. I first came across it in the documentation for Apple's Yellow Box classes.

27.10 Leo's load process

Leo reads local files twice. The first load discovers the settings to be used in the second load. This ensures that proper settings are *available* during the second load.

Ctors init settings ``early", before calling the ctors for subsidiary objects. This ensures that proper settings are *in effect* for the subsidiary ctors.

After creating all subsidiary objects, c.__init__ simply calls c.finishCreate! Creating Commands objects is now completely self-contained. In particular, c.__init__ now creates the fully-inited gui frame. This is a revolution in Leo's startup code! c.__init__ no longer needs a frame argument, a surprisingly important conceptual simplification.

The old code inited Leo windows in several *places* (c.__init__, the end of g.app.newLeoCommanderAndFrame and the end of g.openWithFileName) and in several *phases*, (g.app.newCommanderAndFrame, g.openWithFileName and c.finishCreate.)

The new code has c.__init__ do *all* the work, in one place, and in one phase. This is supremely important for future maintainers. The old code was difficult for me to understand yesterday, even after a full week of study. The new code is a simple as could possibly be imagined. This is a gigantic step forward for Leo.

第 **28** 章

White Papers

I wrote this white paper soon after discovering Python in 2001. The conclusions are still valid today.

Contents

- White Papers
  - Why I like Python
    - Clarity
    - Power
    - Safety
    - Speed
    - Conclusions

28.1 Why I like Python

I've known for a while that Python was interesting; I attended a Python conference last year and added Python support to Leo. But last week I got that Python is something truly remarkable. I wanted to convert Leo from wxWindows to wxPython, so I began work on c2py, a Python script that would help convert from C++ syntax to Python. While doing so, I had an Aha experience. Python is more than an incremental improvement over Smalltalk or C++ or objective-C; it is ``something completely different". The rest of this post tries to explain this difference.

28.1.1 Clarity

What struck me first as I converted C++ code to Python is how much less blah, blah, blah there is in Python. No braces, no stupid semicolons and most importantly, *no declarations*. No more pointless distinctions between const, char *, char const *, char * and wxString. No more wondering whether a variable should be signed, unsigned, short or long.

Declarations add clutter, declarations are never obviously right and declarations don't prevent memory allocation tragedies. Declarations also hinder prototyping. In C++, if I change the type of something I must change all related declarations; this can be a huge and dangerous task. With Python, I can change the type of an object without changing the code at all! It's no accident that Leo's new log pane was created first in Python.

Functions returning tuples are a ``minor" feature with a huge impact on code clarity. No more passing pointers to data, no more defining (and allocating and deallocating) temporary structs to hold multiple values.

Python can't check declarations because there aren't any. However, there is a really nifty tool called Pychecker that does many of the checks typically done by compilers. See pychecker for details.

28.1.2 Power

Python is much more powerful than C++, not because Python has more features, but because Python needs *less* features. Some examples:

- Python does everything that the C++ Standard Template Library (STL) does, without any of the blah, blah, blah needed by STL. No fuss, no muss, no code bloat.
- Python's slicing mechanism is very powerful and applies to any sequence (string, list or tuple). Python's string library does more with far less functions because slices replace many functions typically found in other string libraries.
- Writing dict = {} creates a dictionary (hash table). Hash tables can contain anything, including lists and other hash tables.
- Python's special functions, __init__, __del__, __repr__, __cmp__, etc. are an elegant way to handle any special need that might arise.

28.1.3 Safety

Before using Python I never fully realized how difficult and dangerous memory allocation is in C++. Try doing:

```
aList[i:j] = list(aString)
```

in C. You will write about 20 lines of C code. Any error in this code will create a memory allocation crash or leak.

Python is fundamentally safe. C++ is fundamentally unsafe. When I am using Python I am free from worry and anxiety. When I am using C++ I must be constantly ``on guard." A momentary lapse can create a hard-to-find pointer bug. With Python, almost nothing serious can ever go wrong, so I can work late at night, or after a beer. The Python debugger is always available. If an exception occurs, the debugger/interpreter tells me just what went wrong. I don't have to plan a debugging strategy! Finally, Python recovers from exceptions, so Leo can keep right on going even after a crash!

28.1.4 Speed

Python has almost all the speed of C. Other interpretive environments such as icon and Smalltalk have clarity, power and safety similar to Python. What makes Python unique is its seamless way of making C code look like Python code. Python executes at essentially the speed of C code because most Python modules are written in C. The overhead in calling such modules is negligible. Moreover, if code is too slow, one can always create a C module to do the job.

In fact, Python encourages optimization by moving to higher levels of expression. For example, Leo's Open command reads an XML file. If this command is too slow I can use Python's XML parser module. This will speed up Leo while at the same time raising the level of the code.

28.1.5 Conclusions

Little of Python is completely new. What stands out is the superb engineering judgment evident in Python's design. Python is extremely powerful, yet small, simple and elegant. Python allows me to express my intentions clearly and at the highest possible level.

The only hope of making Leo all it can be is to use the best possible tools. I believe Python will allow me to add, at long last, the new features that Leo should have.

Edward K. Ream, October 25, 2001. P.S., September, 2005:

Four years of experience have only added to my admiration for Python. Leo could not possibly be what it is today without Python.

第 29 章

Appendices

Contents

- Appendices
  - Format of .leo files
  - Format of external files
  - Unicode reference

29.1 Format of .leo files

This technical information may be of use to those wanting to process Leo files with special-purpose filters. Leo's uses XML for its file format. The following sections describe this format in detail. **Important**: The actual read/write code in leoFileCommands.py is the authoritative guide. When in doubt about what Leo actually writes, look at an actual .leo file in another editor.

Here are the elements that may appear in Leo files. These elements must appear in this order.

<?xml> Leo files start with the following line:

```
<?xml version="1.0" encoding="UTF-8"?>
```

<?xml-stylesheet> An xml-stylesheet line is option. For example:

```
<?xml-stylesheet ekr_stylesheet?>
```

<leo_file> The <leo_file> element opens an element that contains the entire file. </leo_file> ends the file.

<leo_header> The <leo_header> element specifies version information and other information that affects how Leo parses the file. For example:

```
<leo_header file_format="2" tnodes="0" max_tnode_index="5725" clone_windows="0"/>
```

The file_format attribute gives the `major' format number. It is `2' for all 4.x versions of Leo. The tnodes and clone_windows attributes are no longer used. The max_tnode_index attribute is the largest tnode index.

<globals> The globals element specifies information relating to the entire file. For example:

```
<globals body_outline_ratio="0.50">
  <global_window_position top="27" left="27" height="472" width="571"/>
  <global_log_window_position top="183" left="446" height="397" width="534"/>
</globals>
```

- The body_outline_ratio attribute specifies the ratio of the height of the body pane to the total height of the Leo window. It initializes the position of the splitter separating the outline pane from the body pane.
- The global_window_position and global_log_window_position elements specify the position of the Leo window and Log window in global coordinates:

<preferences> This element is vestigial. Leo ignores the <preferences> element when reading. Leo writes an empty <preferences> element.

<find_panel_settings> This element is vestigial. Leo ignores the <find_panel_settings> element when reading. Leo writes an empty <find_panel_settings> element.

<clone_windows> This element is vestigial. Leo ignores the <clone_windows> element when reading. Leo no longer writes <clone_windows> elements.

<vnodes> A single <vnodes> element contains nested <v> elements. <v> elements correspond to vnodes. The nesting of <v> elements indicates outline structure in the obvious way.

<v> The <v> element represents a single vnode and has the following form:

```
<v...><vh>sss</vh> (zero or more nested v elements) </v>
```

The <vh> element specifies the headline text. sss is the headline text encoded with the usual XML escapes. As shown above, a <v> element may contain nested <v> elements. This nesting indicates outline structure in the obvious way. Zero or more of the following attributes may appear in <v> elements:

```
t=name.timestamp.n
a="xxx"
```

The t="Tnnn" attribute specifies the <t> element associated with a <v> element. The a="xxx" attribute specifies vnode attributes. The xxx denotes one or more upper-case letters whose meanings are as follows:

```
C    The vnode is a clone. (Not used in 4.x)
E    The vnode is expanded so its children are visible.
M    The vnode is marked.
T    The vnode is the top visible node.
V    The vnode is the current vnode.
```

For example, a="EM" specifies that the vnode is expanded and is marked.

New in 4.0:

- <v> elements corresponding to @file nodes now contain tnodeList attributes. The tnodeList attribute allows Leo to recreate the order in which nodes should appear in the outline. The tnodeList attribute is a list of gnx's: global node indices. See Format of external files (4.x) for the format of gnx's.
- Plugins and scripts may add attributes to <v> and <t> elements. See Writing plugins for details.

<tnodes> A single <tnodes> element contains a non-nested list of <t> elements.

<t> The <t> element represents the body text of the corresponding <v> element. It has this form:

```
<t tx="<gnx>">sss</t>
```

The tx attribute is required. The t attribute of <v> elements refer to this tx attribute. sss is the body text encoded with the usual XML escapes.

New in 4.0: Plugins and scripts may add attributes to <v> and <t> elements. See Writing plugins for details.

29.2 Format of external files

This section describe the format of external files. Leo's sentinel lines are comments, and this section describes those comments.

Files derived from @file use gnx's in @+node sentinels. Such gnx's permanently and uniquely identify nodes. Gnx's have the form:

```
id.yyyymmddhhmmss
id.yyyymmddhhmmss.n
```

The second form is used if two gnx's would otherwise be identical.

- id is a string unique to a developer, e.g., a cvs id.
- yyyymmddhhmmss is the node's creation date.
- n is an integer.

Here are the sentinels used by Leo, in alphabetical order. Unless otherwise noted, the documentation applies to all versions of Leo. In the following discussion, gnx denotes a gnx as described above.

@<< A sentinel of the form @<<section_name>> represents a section reference.

If the reference does not end the line, the sentinel line ending the expansion is followed by the remainder of the reference line. This allows the Read code to recreate the reference line exactly.

@@ The @@ sentinel represents any line starting with @ in body text except @*whitespace*, @doc and @others. Examples:

```
@@nocolor
@@pagewidth 80
@@tabwidth 4
@@code
```

@afterref Marks non-whitespace text appearing after a section references.

@+all Marks the start of text generated by the @all directive.

@-all Marks the end of text generated by the @all directive.

@at and @doc

The @+doc @+at sentinels indicate the start of a doc parts.

We use the following **trailing whitespace convention** to determine where putDocPart has inserted line breaks:

```
A line in a doc part is followed by an inserted newline
if and only if the newline if preceded by whitespace.
```

To make this convention work, Leo's write code deletes the trailing whitespace of all lines that are followed by a ``real'' newline.

@+body (Leo 3.x only) Marks the start of body text.

@-body (Leo 3.x only) Marks the end of body text.

@delims The @delims directive inserts @@delims sentinels into the external file. The new delimiter strings continue in effect until the next @@delims sentinel *in the external file* or until the end of the external file. Adding, deleting or changing @@delim *sentinels* will destroy Leo's ability to read the external file. Mistakes in using the @delims *directives* have no effect on Leo, though such mistakes will thoroughly mess up a external file as far as compilers, HTML renderers, etc. are concerned.

@+leo Marks the start of any external file. This sentinel has the form:

```
<opening_delim>@leo<closing_delim>
```

The read code uses single-line comments if <closing_delim> is empty. The write code generates single-line comments if possible.

The @+leo sentinel contains other information. For example:

```
<opening_delim>@leo-ver=4-thin<closing_delim>
```

@-leo Marks the end of the Leo file. Nothing but whitespace should follow this directive.

@+middle (Leo 4.0 and later) Marks the start of intermediate nodes between the node that references a section and the node that defines the section. Typically no such sentinels are needed: most sections are defined in a direct child of the referencing node.

@-middle (Leo 4.0 and later) Marks the end of intermediate nodes between the node that references a section and the node that defines the section.

@+node Mark the start and end of a node.

> @+node:gnx:<headline>

@others The @+others sentinel indicates the start of the expansion of an @+others directive, which continues until the matching @-others sentinel.

@verbatim @verbatim indicates that the next line of the external file is not a sentinel. This escape convention allows body text to contain lines that would otherwise be considered sentinel lines.

@@verbatimAfterRef @verbatimAfterRef is generated when a comment following a section reference would otherwise be treated as a sentinel. In Python code, an example would be:

```
<< ref >> #+others
```

29.3 Unicode reference

Leo uses unicode internally for all strings.

1. Leo converts headline and body text to unicode when reading .leo files and external files. Both .leo files and external files may specify their encoding. The default is utf-8. If the encoding used in a external file is not ``utf-8'' it is represented in the @+leo sentinel line. For example:

```
#@+leo-encoding=iso-8859-1.
```

   The utf-8 encoding is a ``lossless'' encoding (it can represent all unicode code points), so converting to and from utf-8 plain strings will never cause a problem. When reading or writing a character not in a ``lossy'' encoding, Leo converts such characters to `?' and issues a warning.

2. When writing .leo files and external files Leo uses the same encoding used to read the file, again with utf-8 used as a default.

3. leoSettings.leo contains the following Unicode settings, with the defaults as shown:

```
default_derived_file_encoding = UTF-8
new_leo_file_encoding = UTF-8
```

   These control the default encodings used when writing external files and .leo files. Changing the new_leo_file_encoding setting is not recommended. See the comments in leoSettings.leo. You may set default_derived_file_encoding to anything that makes sense for you.

4. The @encoding directive specifies the encoding used in a external file. You can't mix encodings in a single external file.

第 **30** 章

Glossary

This is a short glossary of important terms in Leo's world. For more information about terms, look in the index for links to discussions in other places, especially in Leo's Tutorial and Leo's Reference.

@ Starts a doc part. Doc parts continue until an @c directive or the end of the body text.

@@ convention for headlines Within @asis trees only, if a headline starts with @@, Leo writes everything in the headline following the @@ just before the corresponding body text.

@<file> node A node whose headline starts with @asis, @edit, @file, @nosent, @shadow, @thin, or their longer forms. We often refer to outline nodes by the directives they contain. For example, an @file node is a node whose headline starts with @file, etc.

@all Copies the body text of all nodes in an @file tree to the external file.

@asis <filename> Creates an external file containing no Leo sentinels directly from the @asis tree.

@auto <filename>

Imports an external file into the Leo outline, splitting the file into pieces if an importer exists for the give filetype. Importers presently exist for the following languages: C, C++, C#, HTML, INI files, Java, PHP, Pascal, Python and XML.

@c and @code

Ends a doc part and starts a code part.

@chapter and @chapters

An @chapter tree represents a chapter. All @chapter nodes should be contained in an @chapters node.

@color

Enables syntax coloring in a node and its descendants until the next @nocolor directive.

@comment

Sets the comment delimiters in @thin, @file and @shadow files.

@delims

Sets the comment delimiters in @thin, @file and @shadow files.

@edit <filename>

Reads an entire external file into a single node.

@encoding <encoding>

Specifies the Unicode encoding for an external file.

@end_raw

Ends a section of `raw' text.

@file <filename>

Creates an external file containing sentinels. When writing @file trees, Leo expands section references and @all and @others directives.

Important: @file is the recommended way to create and edit most files. Using @file trees is **highly recommended** when sharing external files in a collaborative environment.

@first <text>

The @first directive allows you to place one or more lines at the very start of an external file, before the first sentinel. For example:

```
@first #! /usr/bin/env python
```

@killcolor

Completely disables syntax coloring in a node, regardless of other directives.

@language <language name>

Specifies the source language, which affects syntax coloring and the comments delimiters used in external files and syntax coloring.

@last <text>

Allows you to place lines at the very end of external files, after the last sentinel. For example:

```
@first <?php
...
@last ?>
```

@lineending cr/lf/nl/crlf

Sets the line endings for external files.

@nocolor

Disables syntax coloring in a node and its descendants until the next @color directive.

@nocolor-node

Completely disables coloring for one node. Descendant nodes are not affected.

@nosent

Creates an external file containing no sentinels. Unlike @asis, sections references and the @all and @others directives are valid in @nosent trees.

@nowrap

Disables line wrapping the Leo's body pane.

@others

Copies the body text of all nodes *except* section definition nodes in an @file tree to the corresponding external file.

@pagewidth <n>

Sets the page width used to break doc parts into lines.

@path <path>

Sets the path prefix for relative filenames for descendant @<file> directives.

@raw Starts a section of ``raw'' text that ends *only* with the @end_raw directive or the end of the body text.

@tabwidth <n>

Sets the width of tabs. Negative tab widths cause Leo to convert tabs to spaces.

@thin <filename>

A synonym for @file.

@wrap

Enables line wrapping in Leo's body pane.

Body pane

The pane containing the body text of the currently selected headline in the outline pane.

Body text

The text in the body pane. That is, the contents of a node.

Body text box

A small blue box in the icon box indicating that the node contains body text.

Child

The direct descendant of a node.

Clone A copy of a tree that changes whenever the original changes. The original and all clones are treated equally: no special status is given to the ``original'' node.

Clone arrow

A small red arrow in the icon box indicating that the node is a clone.

Code part

A part of a section definition that contains code. Code parts start with @c or @code directives and continue until the next doc part.

Contract:

To hide all descendants of a node.

Demote To move all siblings that follow a node so that they become children of the node.

Descendant An offspring of a node. That is, a child, grandchild, etc. of a node.

Directive

A keyword, preceded by an `@' sign, in body text that controls Leo's operation. The keyword is empty for the @ directive.

Doc part

A part of a section definition that contains comments. Doc parts start with @ and continue until the @c directive or the end of the body text.

Escape convention

A convention for representing sequences of characters that would otherwise have special meaning. **Important**: Leo does not support escape conventions used by noweb. Any line containing matched << and >> is a section reference, regardless of context. To use << and >> as ordinary characters, place them on separate lines.

Expand

To make the children of a node visible.

External file

A file outside of Leo that is connected to Leo by an @<file> node.

Grandchild

The child of a child of a node.

Headline

The headline text of a node. The part of the node visible in the outline pane.

Hoist & dehoist

Hoisting a node redraws the screen that node and its descendants becomes the only visible part of the outline. Leo prevents the you from moving nodes outside the hoisted outline. Dehoisting a node restores the outline. Multiple hoists may be in effect: each dehoist undoes the effect of the immediately preceding hoist.

Icon box

An icon just to the left of headline text of a node indicating whether the node is cloned, marked or dirty, and indicating whether the node contains body text.

Log Pane

The part of Leo's main window that shows informational messages from Leo. It also contains the Find tab, the Spell tab, the autocompletion tab.

Mark

A red vertical line in the icon box of a node.

Node

The organizational unit of an outline. The combination of headline text and body text. Sometimes used as a synonym for tree.

Offspring

A synonym for the descendants of a node. The children, grandchildren, etc. of a node.

Organizer node

A node containing no body text. Organizing nodes may appear anywhere in an @file tree; they do not affect the external file in any way. In particular, organizing nodes do not affect indentation in external files.

Orphan node

A node that would not be copied to a external file. Orphan nodes can arise because an @file tree has no @others or @all directives. Sections that are defined but not used also create orphan nodes.

Leo issues a warning when attempting to write an @file tree containing orphan nodes, and does not save the external file. No information is lost; Leo saves the information in the @file tree in the .leo file. Leo will load the @file tree from the .leo file the next time Leo opens the .leo file.

Outline

A node and its descendants. A tree. All the nodes of a .leo file.

Outline order

The order that nodes appear on the screen when all nodes are expanded.

Outline pane

The pane containing a visual representation of the entire outline, or a part of the outline if the outline is hoisted.

Parent

The node that directly contains a node.

Plugin

A way to modify and extend Leo without changing Leo's core code. See Writing plugins and hooks.

Promote

To move all children of a node in an outline so that they become siblings of the node.

reStructuredText (rST)

A simple, yet powerful markup language for creating .html, or LaTeX output files. See the rST primer.

Root

The first node of a .leo file, outline, suboutline or @<file> tree.

Section

A fragment of text that can be incorporated into external files.

Section definition

The body text of a section definition node.

Section definition node

A node whose headline starts with a section name and whose body text defines a section.

Section name

A name enclosed in << and >>. Section names may contain any characters except newlines and ``>>''.

Section reference

A section name appearing in a code part. When writing to an external file, Leo replaces all references by their definitions.

Sentinel

Comment lines in external files used to represent Leo's outline structure. Such lines start with an @ following the opening comment delimiter. Sentinels embed outline structure into external files.

Do not alter sentinel lines. Doing so can corrupt the outline structure.

Settings:

Plugins and other parts of Leo can get options from @settings trees, outlines whose headline is @settings. When opening a .leo file, Leo looks for @settings trees in the outline being opened and also in various leoSettings.leo files. @settings trees allow plugins to get options without any further support from Leo's core code. For a full discussion of @settings trees, see Customizing Leo.

Sibling

Nodes with the same parent. Siblings of the root node have the hidden root node as their parent.

Target language

The language used to syntax color text. This determines the default comment delimiters used when writing external files.

Tree

An outline. A node and its descendants.

Underindent line

A line of body text that is indented less then the starting line of the class, method or function in which it appears. Leo outlines can not represent such lines exactly: every line in an external file will have at least the indentation of any unindented line of the corresponding node in the outline.

View node

A node that represents a view of an outline. View nodes are typically ordinary, non-cloned nodes that contain cloned descendant nodes. The cloned descendant nodes comprise most of the data of the view. Other non-cloned nodes may add additional information to the view.

第 31 章

What's New in Leo

Contents

- What's New in Leo
  - Leo 4.10
    - New features & commands
    - Plugins
    - Scripts
    - Settings
    - Bugs fixed
  - Previous versions
    - Leo 4.9
    - Leo 4.8
    - Leo 4.7
    - Leo 4.6
    - Leo 4.5
    - Leo 4.4.8
    - Leo 4.4.6
    - Leo 4.4.5
    - Leo 4.4.4
    - Leo 4.4.3
    - Leo 4.4.2
    - Leo 4.4.1
    - Leo 4.4

31.1 Leo 4.10

31.1.1 New features & commands

- Weightless unit testing.
- Added the following commands:

```
beautify-c
c-to-python
clone-find-all-flattened
clone-marked-nodes
delete-marked-nodes
move-marked-nodes
run-marked-unit-tests-externally
run-marked-unit-tests-locally
select-to-matching-bracket
split-defs
```

- Improved the following commands:

```
add-comments
delete-comments
open
page-up
page-down
print-bindings
print-commands
rst3
shell-command
shell-command-on-region
```

- The leoInspect module allows scripts to interrogate static code. For full details, see The leoInspect Module chapter.
- Improved existing features:
  - Leo's File:Open With command now works with Qt
  - The new quick edit/save mode allows Leo to be a drop-in replacement for SciTe.
  - Detached windows.
  - A major improvements to Leo's abbreviation code.
  - Improved presentation of autocompletion list.
  - Applied patch for bug 800399: smart word jumps/deletes.
- Code improvements:
  - Most of Leo's core files now import just leo.core.leoGlobals.
  - Global switches are now all in leoGlobals.py.
  - version.py now uses bzr_version.py. Leo now reports bzr version numbers and dates automatically.

  - Unified the high-level interface & eliminated the low-level interface.
  - Create properties for logCtrl & bodyCtrl.
  - Added event filters to top-level frames.
  - Added g.app.isExternalUnitTest.
  - Added c.config.set.

- Created new classes:

```
EditCommandsManager
KeyStroke & ShortcutInfo
LoadManager
TestManager
```

- Dozens of other new commands and features.

For details, see the release notes.

Weightless unit testing

Small improvements the unit testing framework created big results. They completely eliminate the overhead in running unit tests:

- The run-marked-unit-tests-externally (Alt-4) command runs all marked @test nodes. To choose tests, just mark them.
- Alt-4 now saves the .leo file first.
- **Almost all unit tests may now be run externally:**
  - External unit tests always read config settings.
  - The nullGui now uses a fully capable string-based body widget.
  - The nullGui now uses the regular undoer.
- Unit tests now always have the sources available.
- Disabled messages on external unit tests.
- The unit test commands always run a selected @test node.

These improvements mean that almost all unit tests may be run externally. In turn, this creates a remarkable work flow:

```
Edit
Alt-4
Edit
Alt-4
...
```

The energy difference between weightless and heavy is astounding. Try the new way: you will surely like it.

Notes:

Experience shows that being able to run the desired unit tests *without* selecting any particular node makes an amazingly large difference. Being able to run all and only marked unit tests is a big step forward.

If a marked node is neither an @test node nor an @suite node, all nodes in the tree are considered to be marked.

31.1.2 Plugins

- Improved the quicksearch plugin.
- A new bigdash plugin.
- Removed scrolledmessage plugin.
- The vim and xemacs plugins now work smoothly with contextmenu plugin.
- Supported auto-hide in viewrendered plugin.

31.1.3 Scripts

- Improved the create @auto nodes script.
- Added import-org-mode script.
- Added a script for displaying a function call hierarchy in Leo.
- Improved recursive import script.
- Created a script for replacing Qt stylesheets on the fly.
- Scripts to add bookmarks automatically.
- A new ``magic refactor'' button.
- Changed calling signatures of g.openWithFileName and g.app.newCommander.
- The open-with event now uses a ``d'' arg.

For full details, see the release notes.

31.1.4 Settings

- New settings:
  - @bool indent_added_comments
  - @color focus_border_color = red
  - @int focus_border_width = 1
  - @bool use_body_focus_border = True
  - @bool use_focus_border = True
- Other changes:
  - Added show-decoration-selected: 1 to QTreeWidget stylesheet. This causes the entire headline row to be shown when selected.
  - Added stylesheets for Log & Find tabs.
  - Eliminated the -c option.

  - New format for @openwith settings nodes. See leoSettings.leo for details.

- New search order for leoSettings.leo:

```
1. leoSettings.leo in the home directories.
2. <machine-name>leoSettings.leo in the home directories.
3. leoSettings.leo in leo/config directory.
```

- New search order for myLeoSettings.leo:

```
1. myLeoSettings.leo in the local directory.
2. myLeoSettings.leo in the home directories.
3. <machine-name>myLeoSettings.leo in the home directories.
4. myLeoSettings.leo in leo/config directory.
```

- New default settings for run unit tests commands:

```
run-marked-unit-tests-externally   = Alt-4
run-selected-unit-tests-externally = Alt-5
```

31.1.5 Bugs fixed

- Fixed several bugs related to selection following hoists & chapters:
  - bug 823267: When a tab is closed focus may go to a tab other than the visible one.
  - bug 875327: Positioning outside of hoisted outline" usually causes problems.
  - bug 917814: Switching Log Pane tabs is done incompletely.
  - bug 875323: Hoist an @chapter node leaves a non-visible node selected.
  - bug 831658: @url doesn't leave Chapter.
- Fixed several bugs related to URL's:
  - bug 951739: xdg-open of a file-scheme URL containing blanks.
  - bug 951721: @url with URL in headline.
  - bug 944555: Ctrl-left-click URL handling not as sophisticated as @url URL handling.
  - bug 944551: @url URL Open Hangs Leo.
  - bug 893230: URL coloring does not work for many Internet protocols.
  - Removed ``significant" calls to os.system.
  - Added support for colorizing the following schemes: gopher,mailto,news,nntp,prospero,telnet,wais.
- Fixed several other serious bugs:
  - bug 800157, an ancient hanger in paste-retaining-clones.
  - Fixed at serious read bug. Changed at.readEndOthers and at.readEndRef.
  - Fixed bug: @button @key=x does not override x.
  - Fixed the wretched scrolling bug.

- Fixed almost 70 minor bugs. For details, see the release notes.

31.2 Previous versions

31.2.1 Leo 4.9

Bugs fixed

- Fixed an important bug involving orphan nodes. Leo now never saves an external file containing orphan nodes. This ensures that all the information in the external file will, in fact, be saved in the .leo file.
- Almost 40 minor bugs have been fixed. For details, see the release notes.
- Fixed mod_http plugin

Deprecated/removed features

- Leo no longer supports the Tk gui. The Qt gui now does everything the Tk gui did and better.
- Removed show/hide/toggle minibuffer commands. The minibuffer is an essential part of Leo.
- These settings are no longer used:

```
@string selected-background-color
@string selected-command-background-color
```

- The import-at-root command is no longer supported.

Major improvements

- Support multiple @language directives in a single node As with @color directives, only unambiguous @language directives affect the default coloring of descendant nodes.
- Colorize url's in the body text. You can open url's by control-clicking on them, or by using the open-url command.
- Use @file extension by default if there is no @language directive in effect. This is oh so useful.
- Unified extract commands. This command creates a child node from the selected body text as follows:

    1. If the selection starts with a section reference, the section name become the child's headline. All following lines become the child's body text. The section reference line remains in the original body text.
    2. If the selection looks like a Python class or definition line, the class/function/method name becomes child's headline and all selected lines become the child's body text.
    3. Otherwise, the first line becomes the child's headline, and all selected lines become the child's body text.

    Note that the extract-section-names command remains. The extract-section and extract-python-method commands are gone.

- The import-file commands replaces all the following commands:

```
import-at-file
import-cweb-files
import-derived-file
import-flattened-outline
import-noweb-files
```

  Leo chooses one of the above commands as follows. First, if the file looks like an external file that Leo wrote, the command works like import-derived-file command. Otherwise, the file's extension determines the importer:

```
.cw, .cweb:    import-cweb-files
.nw, .noweb:   import-noweb-files
.txt:          import-flattened-outline
all others:    import-at-file
```

  The import-at-root command is no longer supported.

Completed Leo's autocompleter

Terminology: the *legacy* (aka tabbed) autocompleter shows completions in Leo's tabbed pane. The *new* (aka qcompleter) autocompleter shows completions in or near the body pane.

Appearance There is little change to the legacy completer, except that no text is highlighted in the body pane during completion. This is calmer than before. Furthermore, there is no longer any need for highlighting, because when the user types a backspace the legacy completer now simply deletes a single character instead of the highlighted text.

One minor change: the legacy completer now *does* insert characters that do not match the start of any possible completion. This is an experimental feature, but it might play well with using codewise completions as a fallback to leo-related completions.

Function and design Both the legacy and new completer now work *exactly* the same way, because they both use the AutoCompleterClass to compute the list of completions.

The strict ``stateless" requirement means that the ``intermediate" completions must be entered into the body pane while completion is active. It works well as a visual cue when using the tabbed completer: indeed, the tabbed completer would be difficult to use without this cue.

The situation is slightly different with the qcompleter. Adding code before the user accepts the completion might be considered an ``advanced" feature. However, it does have two important advantages, especially when ``chaining" across periods: it indicates the status of the chaining and it limits what must appear in the qcompleter window.

Codewise completions The codewise-oriented completions appear to work well. In large part, this is due to adding the global ``self." completions to all class-related completions (kind == `class' in ac.get_codewise_completions). This looks like a really good hack, and it eliminates the need for the ContextSniffer class.

Performance Performance of leo-related completions is *much* better than before. The old code used Python's inspect module and was horribly complex. The new code uses eval and is perfectly straightforward.

The present codewise-related code caches completions for all previously-seen prefixes. This dramatically speeds up backspacing. Global caching is possible because completions depend *only* one the present prefix, *not* on the presently selected node. If ContextSniffer were used, completions would depend on the selected node and caching would likely be impractical. Despite these improvements, the performance of codewise-oriented completions is noticeably slower than leo-related completions.

Performance notes The ac.get_cached_options cuts back the prefix until it finds a cached prefix. ac.compute_completion_list then uses this (perhaps-way-too-long-list) as a starting point, and computes the final completion list by calling g.itemsMatchingPrefixInList.

This may not be absolutely the fastest way, but it is much simpler and more robust than attempting to do ``prefix AI'' based on comparing old and new prefixes. Furthermore, this scheme is completely independent of the how completions are actually computed. The autocompleter now caches options lists, regardless of whether using eval or codewise.

In most cases the scheme is extremely fast: calls to get_completions replace calls to g.itemsMatchingPrefixInList. However, for short prefixes, the list that g.g.itemsMatchingPrefixInList scans can have thousands of items. Scanning large lists can't be helped in any case for short prefixes.

Happily, the new scheme is still *completely* stateless: the completionDict does *not* define state (it is valid everywhere) and no state variables had to be added. In short, the new caching scheme is much better than before, and it probably is close to optimal in most situations.

Greatly improved the viewrendered plugin

The viewrendered plugin creates a window for *live* rendering of images, movies, sounds, rst, html, etc.

Commands viewrendered.py creates the following (Alt-X) commands:

viewrendered (abbreviated vr) Opens a new rendering window.

By default, the rendering pane renders body text as reStructuredText, with all Leo directives removed. However, if the body text starts with < (after removing directives), the body text is rendered as html.

Important: The default rendering just described does not apply to nodes whose headlines begin with @image, @html, @movie, @networkx, @svg and @url. See the section called **Special Renderings** below.

Rendering sets the process current directory (os.chdir()) to the path to the node being rendered, to allow relative paths to work in .. image:: directives.

hide-rendering-pane Makes the rendering pane invisible, but does not destroy it.

lock-unlock-rendering-pane Toggles the locked state of the rendering pane. When unlocked (the initial state), the rendering pane renders the contents of the presently selected node. When locked, the rendering pane does not change when other nodes are selected. This is useful for playing movies in the rendering pane.

pause-play-movie This command has effect only if the rendering pane is presently showing a movie. It pauses the movie if playing, or resumes the movie if paused.

show-rendering-pane Makes the rendering pane visible.

toggle-rendering-pane Shows the rendering pane if invisible, otherwise hides it.

update-rendering-pane Forces an update of the rendering pane. This is especially useful for @graphics-script nodes: such nodes are update automatically only when selected, not when the body text changes.

Rendering reStructuredText For example, both:

```
Heading
-------

`This` is **really** a line of text.
```

and:

```
<h1>Heading<h1>

<tt>This</tt> is <b>really</b> a line of text.
```

will look something like:

> **Heading**
>
> *This* is **really** a line of text.

Important: reStructuredText errors and warnings will appear in red in the rendering pane.

Special renderings This plugin renders @image, @html, @movie, @networkx, @svg and @url nodes in special ways.

For @image, @movie and @svg nodes, either the headline or the first line of body text may contain a filename. If relative, the filename is resolved relative to Leo's load directory.

- @graphics-script executes the script in the body text in a context containing two predefined variables:
  - gs is the QGraphicsScene for the rendering pane.
  - gv is the QGraphicsView for the rendering pane.

  Using these variables, the script in the body text may create graphics to the rendering pane.
- @image renders the file as an image.
- @html renders the body text as html.
- @movie plays the file as a movie. @movie also works for music files.
- @networkx is non-functional at present. It is intended to render the body text as a networkx graph. See http://networkx.lanl.gov/
- @svg renders the file as a (possibly animated!) svg (Scalable Vector Image). See http://en.wikipedia.org/wiki/Scalable_Vector_Graphics **Note**: if the first character of the body text is < after removing Leo directives, the contents of body pane is taken to be an svg image.
- @url is non-functional at present.

Settings

- @color rendering-pane-background-color = white The background color the rendering pane when rendering text.
- @bool view-rendered-auto-create = False When True, show the rendering pane when Leo opens an outline.
- @bool view-rendered-auto-hide = False When True, hide the rendering pane for text-only renderings.
- @string view-rendered-default-kind = rst The default kind of rendering. One of (big,rst,html)
- @bool scrolledmessage_use_viewrendered = True When True the scrolledmessage dialog will use the rendering pane, creating it as needed. In particular, the plugins_menu plugin will show plugin docstrings in the rendering pane.

Acknowledgment Terry Brown created this initial version of this plugin, and the free_layout and NestedSplitter plugins used by viewrendered.

New and improved features

Colorizing

- Support multiple @language directives in a single node As with @color directives, only unambiguous @language directives affect the default coloring of descendant nodes.
- Colorize url's in the body text. You can open url's by control-clicking on them, or by using the open-url command.
- Added support for cython colorizing
- Leo ignores (and does not color) @language directive for unknown languages.
- Leo completely recolors nodes when you change @language directives by typing.

Command-line arguments & settings

- The --no-splash command-line option suppresses the splash screen. Leo puts up no splash screen when the --silent or --script command-line options are given.
- Added @bool view-rendered-auto-create setting.
- Added @bool use_qcompleter setting.
- Added auto_tab_complete setting.
- Removed @bool use_codewise setting.
- You now may set icon button colors in the Qt stylesheet.

File handling

- Added namespace and Leo comment lines to .leo files
- Leo opens leoSettings.leo only once
- Fixed Bug 745824: @doc duplicates comment delims in html files https://bugs.launchpad.net/leo-editor/+bug/745824e

- Leo no longer wraps @doc lines. This ensures that Leo does not change files unnecessarily.

Gui

- If you type a *plain* up/down arrow key while editing a headline, Leo will act as if you had typed the corresponding *alt-* arrow key. That is, Leo will end editing of the headline and go to the next previous node. Leo will end editing even if there is no next/ previous node, which is convenient.
- A single click on an already-selected tree node edits the headline

  Enabled only if @bool single_click_auto_edits_headline = True.
- Added a splash screen

  The --no-splash command-line option suppresses the splash screen. In addition, Leo puts up no splash screen when the --silent or --script command-line options are given. To change the splash screen, replace leoIconsSplashScreen.jpg with another image.
- The apropos commands now print in a separate area if possible. The commands use the scrolledmessage plugin if possible, which in turn uses the viewrendered plugin by default. This makes the apropos messages much more visible.
- Handle click events like alt-x or ctrl-g Clicking in the minibuffer now is equivalent to alt-x, provided that the minibuffer is not in use. Clicking most places outside the minibuffer is equivalent to ctrl-g. Catching clicks is much safer than catching focus events.
- The first loaded file sets tabbed gui size
- Enter insert mode after ctrl-h. This is a vim-related improvement.
- Disabled find/change text areas in find panel. This reduces confusion.

Improved commands

- Improved the clone-find-all command. The descendants of previously found (cloned) nodes don't get added again. The clone-find-all pattern now defaults to find text.
- Improved the forward and backward by sentences commands Leo's sentence related functions now stop at empty lines, skip periods within words, stop at sentences ending in non-periods and stop at the end or beginning of the buffer.
- Improved the print-bindings command; it now shows were bindings came from.
- Improved the reformat-paragraph command. The command detects paragraphs more reliably. The next line is now visible, which is a big improvement.
- Added patch to g.wrap_lines from Josjas Echenique It regularizes the number of spaces after periods.
- Improved expansion of abbreviations. Abbreviations are checked any time a non-word character is typed. In particular, newlines trigger abbreviations, which I find very helpful, although I did then have to remove newlines from my abbreviations. Control sequences do not trigger expansions.
- Improved handling of @url nodes. The new rule is simple: if the body text contains any text the first line of the body text is taken to be the url. There is no longer any need to put `--` in the headline. More importantly, you can put anything you like in the body text following the first line. Other url's, notes, even .. graphics:: directives for the viewrendered plugin.

- Improved the clean-all-lines command. It is now much faster and has better feedback.

New commands

- Added the replace-current-character command. It replaces the character to the left of the cursor, or replaces the selection range if there is one.
- Added toggle-case-region command.
- Added save-all command. It saves all changed windows.
- Added insert-hard/soft-tab commands.
- Added commands to manage uA's:

```
clear-all-uas
clear-node-uas
print-all-uas
print-node-uas
set-ua
```

- Renamed the `abbrev-mode' to `toggle-abbrev-mode'.

Scripting

- Added namespace arg in c.executeScript
- Put Kent Tenney's Runwith class in scripts.leo and contrib.

  Kent writes, ``I've had endless problems with interpreter versioning, leading me create the Runwith class. It writes a file to disk, makes it executable, runs it, captures exitcode, err and output, removes the files, provides reports. This provides complete decoupling from Leo."
- Call os.chdir when executing scripts.

New in 4.9 b2

- Double-clicking a headline now colorizers the headline exactly the same way as when editing the headline with ctrl-H. This was a serious problem for those with dark window-color schemes.
- The distribution script now ensures that leopluginsspellpyx.txt contains Linux-style newlines. This prevents crashes in the PyEnchant spell checker.
- Leo imports .cfg files just like .ini files.
- Fixed crasher in graphcanvas plugin caused by a bug in CommandChainDispatcher.add.

New in 4.9 b3

4.9 b3: Bugs fixed

- Fixed ancient, major bug: F3 now makes sure to save headline changes

- Fixed old bug: set-find-x commands no longer abort find commands

  The commands that switch find scope, set-find-xxx, no longer terminate the find command, if one is active. This is an old bug, and it's good to fix it.

- Fixed recent bugs in the viewrendered and scrolledmessage plugins

  An earlier rev fixed a bug that effectively destroyed the viewrendered plugin. It was caused by the new convention that alleviates the need for many @language directives. The fix was simply to enable the update_rst method if the massaged p.b is not empty.

  ScrolledMessageDialog.convertMessage now renders rst by default, unless *either* the html or text button is pressed. There really should be three radio buttons: text, html or rST, but that's a tiny interface glitch. The actual bug however, was much more serious: rst was never being rendered.

- Fixed chapters problems

  > http://groups.google.com/group/leo-editor/browse_thread/thread/3f15a855ca38b26e
  >
  > The new code is more relaxed about where @chapter nodes may reside. They are always *created* as the last child of the first @chapters node in the outline (the @chapters, plural, node is created as needed). However, you may move them while in the ``main'' chapter, with no ill effects. In fact, you could swap @chapter nodes with the same name: when you select a chapter, Leo will use (show) the first node it finds.
  >
  > The new code is now both more careful and more tolerant of @chapter nodes deleted by hand. The chapter will still appear in the dropdown list: if you select it you will give a polite warning. That's all. In particular, the deleted chapter will *remain* in the dropdown list until you use the proper chapter-remove command. That's about the only sane alternative: it allows you to resurrect the chapter, by hand or with an undo.
  >
  > This is all made possible because the new code is almost completely stateless. The only exception is the saved position used to select a node when selecting a chapter. The old position-based findPositionInChapter method has been simplified to make it work more reliably. It first looks for a ``perfect'' match using positions, and then degrades to looking for a vnode match. In practice, most matches are, in fact, perfect. The ``imperfect'' case typically happens when the user alters nodes in @chapter trees by hand in the ``main'' chapters.
  >
  > Technical highlights:
  >
  > - The check for c.positionExists(p) in c.setCurrentPosition continues to fail when deleting @chapter nodes. However, the code now simply falls back to c.rootPosition, without any apparent harm.
  > - The chapterController and chapter classes are now completely stateless, except for chapter.p.
  >
  >   1. chapter.findPositionInChapter has been simplified and generalized. It now falls back to a reasonable value, based on p.v, if chapter.p does not exist.
  >   2. All chapterController code now recomputes the location of @chapters and @chapter nodes whenever those locations are needed.
  >   3. All chapter commands are unchanged in their actual workings, but all contain a care ``preamble'' of checking code.

- Added unit test for all chapter commands. All interactive commands now have an xByName helper for use by unit tests.

- Added lockout to leoQtTreeTab. This prevents flash during the rename chapter command.
- Rewrote chapter.chapterSelectHelper. This reduces, but does not eliminate, the number of warnings given by c.setCurrentPosition.
- Fixed recent bug: handle `Escape' character properly

  The fix was a last-minute adjustment in leoQtEventFilter.create_key_event.
- Fixed caps-lock problem

  The fix was yet another last-minute fix leoQtEventFiler.create_key_event.
- Made sure all keys contribute to lossage

4.9 b3: New features

- Simplified Leo's key handling, an important improvement to Leo's core.
- Changed names of commands so they have common prefixes

  Any custom key bindings (none are bound by default) will have to change.

  The new prefixes are:

  ```
  abbrev-     abbreviation commands
  buffer-     buffer command
  directory-  director commands
  file-       file commands
  gc-         garbage collection
  macro-      macro expansion
  rectangle-  rectangle commands
  register    register commands
  ```

  The already existing prefixes are:

  ```
  apropos-    help
  ddabrev-    dynamic abbreviations
  find-       find commands
  isearch-    incremental search
  print-      print information
  run-        run unit tests
  toggle-     toggle settings
  yank-       yank
  ```

- Finished macros

  The macro-load and macro-save are as simple as possible.

  No further work will be done on macros unless somebody really wants these commands.
- Added support for word-only option for regular expressions

When the word-only option is in effect, Leo ensures that the search pattern begins and ends with the `b' anchor.

- Leo's startup code now forces the qt gui: it changes qttabs to qt.

- Added support for expanded sections in plugin. Added three new options:

  expand_noweb_references

    True: Replace references by definitions. Definitions must be descendants of the referencing node.

  ignore_noweb_definitions

    True: ignore section definition nodes.

  expand_noweb_recursively

    True: recursively expand definitions by expanding any references found in definitions.

New in 4.9 b4

- Running all unit tests leaves all files unchanged. This was a major annoyance.
- Leo now does a keyboard-quit when deactivating a window.
- Fixed an ancient bug: everything after @all was put in the wrong node!
- Fixed an ancient bug: wrap-around search now restarts when find pattern changes.
- Fixed an ancient bug: F-keys end incremental searches.
- Fixed a serious recent problem with commands dispatched from menus The Shift modifier was deleted from all commands executed by selecting an item in menus! A new unit test checks that menus behave as expected.
- Dismiss splash screen before putting up the dialog that asks for an ID.

New in 4.9 rc1

- When running on MacOS, Leo uses the qt gui when the qttabs gui is requested.
- Leo now looks in home/.leo/ Icons directory for icons before looking in the leo/ Icons directory. http://groups.google.com/group/leo-editor/browse_thread/thread/80163aec96b8ea45/4f58418924172252
- Fixed bug 797470: File data sometimes silently erased when the tangler fails. https://bugs.launchpad.net/leo-editor/+bug/797470 This was a serious bug, but it could happen only when saving an erroneous file twice.
- Fixed bug 798194: --maximized has no effect https://bugs.launchpad.net/leo-editor/+bug/798194
- Added the @bool forbid_invalid_completions setting.
- Non-plain keys, such as Ctrl-s, abort auto-completion and are interpreted as usual.
- Don't mark the .leo file as changed when setting orphan bit. There is no need: the orphan bits will ensure errors get reported if the file is saved.
- Disabled the open-compare-window command. It is/was a Tk only command.
- The open-python-window command fails more gracefully It issues a message instead of crashing if idlelib does not exist.

31.2.2 Leo 4.8

New sentinels

Leo now writes @file files with the simplest possible sentinel lines.

- Eliminated @-node sentinels.
- Eliminated @nl and @nonl sentinels.
- Simpler representation of @doc and @ in sentinels.
- Simplified representation of @others and section references.
- Use a scheme much like Emacs org-mode to represent headline level.

The result is, provably, the simplest possible representation of Leo's outline structure in external files.

Drag and drop files into Leo

The Qt Gui now supports drag and drop in Leo outlines.

You can drag files into Leo. Leo will create @file or @auto nodes if appropriate.

Scripting improvements

- The execute-script now calls execfile (or its equivalent when using Python 3k) when @bool write_script_file = True. This allows pdb (or pudb) to show the text of Leo scripts!
- Added p.deletePositionsInList, an important new helper.
- Added g.findTestScript, an important new pattern for sharing code in Leo scripts, including scripts in @test nodes.

  Suppose there is common code that I want to include in several unit tests:

  ```
  class Hello():
    def __init__(self,name='john'):
      self.name=name
      print('hello %s' % name)
  ```

  I put this in a node called `Common test code'. Now the unit tests can ``import'' the code as follows:

  ```
  exec(g.findTestScript('Common test code'))
  ```

  After this exec statement completes the class Hello is available to the test code! This is something that I've wanted to do forever.

Improved @url nodes

If the body text is non-empty, it is assumed to contain the URL. This is a remarkably important improvement--it allows the headline to contain a description of the url.

New & improved commands

- Added code-to-rst command.
- Completed cascade-windows and minimize-all-windows commands.
- Created head-to-prev-node and tail-to-next-node commands.
- Removed mark-clones command. It is useless in the one-node world.
- Added extract-python-method command.

Improved abbreviations commands

When abbreviation mode is on (abbrev-mode toggles this mode) Leo will expand abbreviations as you type. Type the name of an abbreviation, followed by a space. As soon as you type the space, Leo will replace the name by the abbreviations value. You can undo the replacement as usual.

Note that defining any abbreviation automatically turns on abbreviation mode.

The add-global-abbreviation command (<alt-x>add-gl<tab><return>) takes the selected text as the replacement value of the abbreviation. The minibuffer prompts you for the name of the abbreviation.

Three new settings apply to the abbreviation commands:

- @bool enable-abbreviations (default: False)

  When true, enables substitution of abbreviations.

- @data global-abbreviations
- @data abbreviations

  In both cases, body text contains lines of the form:

  ```
  name=value
  ```

  name is the abbreviation name, value is the substituted text. Whitespace is ignore around the name, but is significant in the value. Abbreviation names may contain only alphabetic characters, but may start with the `@' sign.

  By *convention* @data global-abbreviations setting should be defined in myLeoSettings.leo, while @data abbreviations should be defined in other .leo files. Regardless of where they are defined, abbreviations in @data abbreviation nodes will override settings (with the same name) in @data global-abbreviations nodes.

New plugins

- The screenshots.py plugin helps make slide shows containing many screen shots.

New settings & command-line args

- Leo can now open multiple files from the command line.

- **You can now set a proportional font to use in all ``@language plain" nodes.** Specify fonts in @font nodes:

```
        @font plain null font

            plain_null_font_family = Times New Roman
            plain_null_font_size = 16
            plain_null_font_slant = roman
            plain_null_font_weight = bold
```

  That is, the actual font specs are in the body text. Everything except @font is ignored in the headline.

  Specify font colors with @color nodes:

```
@color plain null color = black
```

- Added support for minibuffer colors. Added the following options with the indicated defaults:

```
@color minibuffer_background_color = lightblue
@color minibuffer_warning_color = lightgrey
```

- Added support for @string qt-toolbar-location = <spot>

    Valid values for <spot> are top,bottom,left,right

- Added support for @bool write_expansion_bits_in_leo_files.
- The -screen-shot command-line argument tells Leo to take a screenshot and exit.
- The --window-size command-line argument specifies the initial size of the Leo window. This is especially useful with the screen-shot command-line argument:

```
--window-size=600x900  # <height> x <width>, in pixels.
```

- Added support for @bool at_auto_separate_non_def_nodes option.

    When true, the @auto file importers put inter-def code in their own node. The default (legacy mode) is False.

Other improvements

- Several important improvements to Leo's installer for Windows.
- Leo doesn't create @chapter nodes for new files.
- Leo now uses PyEnchant to check spelling.

    This is much safer than the old Aspell wrapper.

- All @auto nodes end with a newline.
- Leo now writes @edit nodes like @nosent nodes.
- Added legend for print-settings command.
- Improved the importer for elisp.
- Added an .ini importer.

- Created introductory slide shows.
- Reorganized the users guide.
- Improved the installation instructions.
- Added support for .nsi files.

31.2.3 Leo 4.7

The one-node world

Leo 4.7 accomplishes something I long thought to be impossible: the unification of vnodes and tnodes. tnodes now longer exist: vnodes contain all data. The Aha that made this possible is that iterators and positions allow a single node to appear in more than one place in a tree traversal.

Leo supports Python 3.x

Leo requires Python 2.6 or above, including Python 3.0 and above.

Improved file handling

- Leo now treats @file nodes just like it treats @thin nodes. This makes Leo much safer to use in cooperative environments that use source code control systems. As part of this change, Leo no longer supports @noref nodes.
- @auto-rst now works much more reliably.
- Leo now has a simple, robust, and extremely useful scheme to recover from clone conflicts, no matter how they may arise. This removes all the dread from ``node changed" messages. It is easy to see what the changes were, and it is easy to choose what, if anything to do.

  When a clone conflict occurs, you will see a red message in the log pane and a ``Recovered Nodes" node as the last top-level node. This node has one child per red message. Each of these children contains two nodes: an ``old" node and a ``new" node. Unless there are multiple conflicts for a single node, the ``new" node will have ``won": every clone contains the new node's headline and body text. All these nodes are plain nodes, *not* clones. It is up to you to change the corresponding clone nodes if you choose to do so.
- Leo minimizes unnecessary changes to .leo files. Leo writes outline-size and orientation to the cache in your .leo directory. This eliminates unnecessary changes to .leo files.
- Leo now creates temporary files in the systems standard temporary directory. This prevents Leo from over-writing user-generated .bak files.

New command-line options

- The --debug command-line option sets g.debug.
- The --version command-line option causes Leo to print it's version and exit.

New commands

- The clear-cache and clear-all-caches commands.

New settings

The qt colorizer now supports font specifications in @font nodes.

Improved plugins

Added options for vim plugin. The setting:

```
@string vim_trigger_event = icondclick2
```

is the default. It opens vim when the user double-clicks the icon box. Alternatives are:

```
@string vim_trigger_event = iconclick2
@string vim_trigger_event = select2
```

The former opens vim on single clicks in the icon bar. The latter opens vim whenever a new node is selected in Leo.

31.2.4 Leo 4.6

Improved unit testing

- leoDynamicTest.py now supports a --path argument giving the .leo file. This is so useful!
- leoDynamicTest.py now honors the --silent argument.
- leoTest.runUnitTestLeoFile runs all unit tests in a given .leo file in a separate process.
- leoTest.runTestsExternally calls runUnitTestLeoFile after creating dynamicUnitTest.leo.
- When reporting that no unit tests were found, all unit tests commands tell whether the entire outline or just the selected outline was searched. This fixes sometimes-misleading error messages.
- test.leo contains a `run-test.leo-tests' button.
- leoPy.leo contains a `run-all-core-tests' button.

Improved file handling

- Leo opens a default .leo file if no other is specified, using the @string default_leo_file setting. The default for this setting is:

  ```
  ~/.leo/workbook.leo
  ```

- Added escapes for underindented lines. The escape is specified by the @string underindent-escape-string setting. By default, this escape is - If a line starts with -N, Leo will write the line with N fewer spaces than expected.
- Leo now warns when attempting to write a file that has been changed outside of Leo. This prevents bzr reversions.

- Leo tests syntax of .py files when saving them.
- Leo can now open any file into an @edit node. This allows Leo to be associated with the edit action of .py files. Like this:

```
C:\Python26\python.exe "c:\leo.repo\trunk\launchLeo.py" --gui=qt %1 %2
```

- Leo now warns if when writing an @auto node if the the file exists and the node has not been read previously. This prevents a newly-created @auto node from overwriting an existing file.

Improved handling of rST files

Added support for @auto-rst nodes. These import reStructuredText (rST) files so that the files can be ``round-tripped'' without introducing extraneous changes. This makes Leo a superb environment for using rST.

New code features

- Added autoCompleter.getExternalCompletions.
- Added g.posList.
- c.config.doEnabledPlugins sets g.app.config.enabledPluginsFileName
- **Added the following properties:**
    - p.b, t.b and v.b return the body string of the position or node.
    - p.h, t.h and v.h return the head string of the position or node.
    - t.u and v.u return the uA of the node.
    - p.gnx, t.gnx and v.gnx return the gnx of the position or node.
- Added script to leoSettings.leo to ensure all menu items are valid.
- c.config.getSettingSource(setting_name) returns the name of the file which Leo used to determine the setting:
    - D indicates default settings.
    - F indicates the file being loaded
    - L indicates leoSettings.leo
    - M indicates myLeoSettings.leo
- Predefined `self' in @test/@suite nodes.
- Added c.getNodePath and c.getNodeFileName.

New command-line options

- The --config command-line option specifies a single config (.leo) file to use for configuration. See http://groups.google.com/group/leo-editor/browse_thread/thread/f3f95d93bcd93b94
- The --file=fileName command-line option loads a file. Only .zip and .leo extensions are allowed at present.
- The --gui=name command-line option specifies the gui to use. The valid values are --gui=qt and --gui=tk.

New commands

- Added smart home (back-to-home) command.
- Added support for standard behavior of Tab and Shift-Tab keys. The tab key indents the text selection, if there is one; otherwise, the tab key insert a tab or blanks, depending on the @tabwidth setting. Shift-Tab always unindents one or more lines.
- The open command creates @edit nodes when opening non-.leo files The open file dialog now shows all files by default. Selecting a non-.leo file will load that file into a new node in the present outline.
- Added added pdb minibuffer command. This works, but stops Leo in the middle of the command-handling logic. You may get the commander c by stepping out into k.masterKeyHandler or k.masterCommandHandler. Using c, you can then get all other info.
- Improved the isearch commands.
- find-clone-all is a synonym for clone-find-all.
- open-quickstart-leo command opens leo/doc/quickstart.leo.
- The Alt-Right and Alt-Left keys (expand-and-go-right and contract-or-go-left commands) now move to the previous or next node if now left/right movement is possible.

New and improved directives

- Added @nocolor-node directive.
- Improved @path handling.

New settings

- @string default_leo_file = ~/.leo/workbook.leo
- @string underindent-escape-string = -
- @int icon_bar_widgets_per_row
- Added support for meta keys.
- The qt gui is now the default.
- The old bindings bound the PageUp/Down keys to back/forward page commands, and these commands work only for text.

  The new default bindings in leoSettings.leo: @keys EKR bindings are:

```
back-page                      ! text = PageUp
back-page-extend-selection     ! text = Shift-PageUp
forward-page                   ! text = PageDn
forward-page-extend-selection  ! text = Shift-PageDn

scroll-down-half-page  ! tree = Shift-PageDn
scroll-down-page       ! tree = PageDn
```

```
scroll-up-half-page    ! tree = Shift-PageUp
scroll-up-page         ! tree = PageUp
```

- @bool enable_alt_ctrl_bindings. The default is False, needed for AltGr functionality on Windows.

Plugins

- Improved nav_buttons plugin and corresponding nodeHistory class.
- Created qtGui and tkGui plugins.
- Created leoGuiPluginsRef.leo.
- Leo issues an error message if a non-existent plugin appears in an @enabled-plugin node.
- New plugins: spydershell.py, qtframecommands.py, and mod_framesize.py.

31.2.5 Leo 4.5

Major new features

- Added support for @shadow files. This is a major breakthrough. See the Using @shadow chapter for full details.
- Added much improved support for vim bindings.
- Allow v.uA's in @file and @shadow nodes.

Major code reorganizations

- Leo now uses a sax-based parser to read .leo files. This makes it possible to extend Leo's file format without invalidating previous versions of Leo.
- Leo now supports the so-called `Graph World'. When g.unified_nodes is True, Leo moves all information from tnodes into vnodes.
- Leo now uses a new key binding scheme. This allows substantially simpler key bindings. Indeed, most per-pane bindings have been eliminated. Added support for kill bindings.
- Leo is now an installable package. To make this work, Leo adds os.curdir to sys.path if needed on startup.
- Reorganized Leo's drawing and focus code. As a result, calls to c.beginUpdate and c.endUpdate are no longer needed.
- Leo is now ready for Python 3.x: Change most print statements to calls to g.pr.

Minor new features

- Added g.Tracer class. This is a Python `debugger' that computes a call graph. To trace a function and its callers, put the following at the function's start:

```
g.startTracer()
```

- The find-character command now finds characters across line boundaries.

- Set cwd in read/write commands. This affect the following commands: open, save, save-as, save-to, read-outline-only, read-file-into-node, write-file-from-node and all the import/export commands.
- Leo creates the .leo folder in the user's HOME directory, and puts several configuration files there. Leo looks for myLeoSettings.leo in HOME/.leo. Leo uses os.path.expanduser(``~'') if there is no home setting.

New settings

- The default settings for @shadow files are now located in leoSettings.leo in the node:

```
    @settings-->File options-->Shadow files

The defaults for these settings are::

    @string shadow_prefix = x
    @string shadow_subdir = .leo_shadow
```

- Added support for @bool fixedWindow option.

  Leo suppresses marks, expansion state, orphan bits and current position bits when writing fixed .leo files. As a result, all nodes will be collapsed and the root node will always be selected when Leo opens a fixed .leo file.

  You can optionally specify the size and position on the screen of fixed .leo files by putting an `@data fixedWindowPosition' node in the @settings tree of myLeoSettings.leo or leoSettings.leo. You should **not** put such a node in the fixed .leo file itself--everyone who opens the file would get that fixed position.

  The body of the `@data fixedWindowPosition' node should contain something like this:

```
# Must be four entries: width,height,left,top.
# Put this in myLeoSettings.leo, **not** in individual .leo files.

1200
800
50
50
```

- Added @bool cleo_color_ignore = True

  This determines whether cleo colors @ignore headlines. The default is True.

31.2.6 Leo 4.4.8

New features

- Better support for unicode in @auto trees.
- All import commands now honor @path
- Leo now supports arguments to minibuffer commands.

- Leo can now translate messages sent to Leo's log. Rather than using an `_' function to denote strings to be translated, Leo's g.es and g.es_print functions translate ``odd'' (first, third, fifth) arguments, leaving ``even'' arguments untranslated. Keyword arguments, color, newline, etc. are never translated. g.translateString does the actual translation using Python's gettext module.
- @menu items may not refer to commands created by @button and @command nodes.

New and improved plugins

- The ipython plugin creates a simple, powerful, effective bridge between IPython and Leo. See http://webpages.charter.net/edreamleo/IPythonBridge.html
- Improved marks/recent buttons plugin.

New settings

- Added support for @commands trees in leoSettings files.
- Added support for @bool open_with_save_on_update setting. If True, Leo will automatically save the outline whenever an external editor changes the outline.

31.2.7 Leo 4.4.6

New commands

```
find-next-clone
toggle-sparse-move
```

Replaced the delete-all-icons command with a script in scripts.leo. This command was too dangerous.

New features

- Added support for @auto xml and @auto javascript. Use @data import_xml_tags setting to specify the xml tags that act as organizers. Javascript regexps that look like section references cause problems, but that can not be helped.

New settings

- Added support for @data nodes in settings files.
- The @data import_xml_tags setting specifies the xml tags that act as organizers. This settings is used by @auto when importing xml files.

31.2.8 Leo 4.4.5

Bug fixed

- Fixed hung (zombie) windows.

- Fixed resurrected (vampire) nodes.

New features

- Leo now supports all directives in headlines.
- Moved all unit tests to unitTest.leo and reorganized the unit tests by Leo source file.
- Installed small icon set from Tango library.
- The rst3 plugin now supports @rst-preformat nodes.

New commands

```
delete-all-icons
delete-first-icon
delete-last-icon
delete-node-icons
insert-icon
reverse-sort-lines
reverse-sort-lines-ignoring-case.
sort-lines-ignoring-case
toggle-collapse_nodes_during_finds
```

New settings

- @bool at_auto_warns_about_leading_whitespace

  This option has effect only when importing so-called non-strict languages, for which leading whitespace is not terribly significant.

- @bool warn_when_plugins_fail_to_load

  There is also an @bool trace_plugins setting.

- @bool vim_plugin_opens_url_nodes

  vim.py does not open url nodes if this setting is False.

31.2.9 Leo 4.4.4

Leo 4.4.4 contains many important features originally planned for later releases. The highlights of Leo 4.4.4:

- **The Great Graph Aha**: A Leo outline doesn't have to *be* an arbitrary graph in order to *represent* an arbitrary graph.

  That is, simple scripts allow Leo outlines to represent arbitrary directed graphs. There is no need for a separate `graph world'. The graphed.py plugin is a direct result of this Aha. It allows you to create general graphs from Leo outlines.

- Support for **@auto nodes**. Such nodes allow people to collaborate using Leo without inserting Leo sentinels in the files Leo generates.

- **@menus trees** in settings files create all of Leo's menus. It is now dead easy to make Leo's menus look the way you want.
- **@buttons trees** in settings files create common @button nodes created in all Leo outlines.
- A new, faster, **colorizer plugin** replaces the __jEdit_colorizer__ plugin.
- New commands for **resolving cvs conflicts**.
- Leo's core is now compatible with jython.

The Great Graph Aha

The Great Graph Aha is:

A Leo outline doesn't have to *be* an arbitrary graph in order to *represent* an arbitrary graph.

So the graph world is unnecessary because we can use Leo nodes and trees as data to other graphing packages.** That is, Python scripts can build arbitrary graphs using Leo's existing nodes and trees. And Python scripts can manipulate those graphs. And Python scripts could do the reverse: manipulate the Leo outline by traversing general graphs. So there is no need to complicate Leo's fundamental data structures. Hurray! Instead, we build on the strengths of already existing graphing packages.

The Great Graph Aha created the opportunity for immediate action:

1. test.leo contains the essential scripts to implement graphs in Leo files. These short, simple, self-contained, easily modifiable scripts make possible everything ever envisaged by the (now-defunct) graph world project:

```
leo2graph: convert a normal Leo tree to a NetworkX graph.
at-graph2graph: convert an @graph tree to a NetworkX graph.
at-networkx2graph: convert an @networkx tree to a NetworkX graph
at-networkx2at-graph: create an @graph tree from an @networkx tree.
```

2. The graphed plugin allows users to manipulate parts of Leo outlines as if they were general graphs. It is still early days for this exciting plugin.

Added support for @auto files

What @auto does

@auto trees allows people to use Leo in collaborative environments without using sentinels in the files Leo generates. In contrast to @nosent, @auto trees can change when the corresponding file changes outside of Leo.

Leo will automatically recreate (import) all @auto trees when reading a .leo file, and will write all dirty @auto trees when saving a .leo file. There are two exceptions to this statement:

1. Leo will never read (import) or write an @auto tree if the root @auto tree is under the influence of an @ignore directive.

2. Saving a .leo file does not save @auto nodes if a) they haven't been changed or b) they do not contain a **significant** amount of information. An @auto tree contains a significant amount of information if it has children or if the root node contains more than 10 characters.

Leo creates @auto trees by parsing the corresponding external file. Parsers create descendant nodes of the @auto tree: one node for each class, method and function in the external file.

Parsers presently exist for C, elisp, Java, Pascal, PHP and Python. Leo determines the language using the file's extension. If no parser exists for a language, the entire body of an @auto tree contains a significant amount of information if it has any children or if the root node contains more than 10 non-blank lines. the external file is copied to the body of the @auto node.

Leo does not write the contents of @auto trees to .leo files. In this respect, @auto trees work much like @file trees. @auto trees whose root node is under the scope of an @ignore directive *will* be written to the .leo, just like @file trees.

Perfect import checks

Leo performs several checks to ensure that the result of importing an external file will be equivalent to the file that writing the @auto tree would produce.

These checks can produces **errors** or **warnings**. Errors indicate a potentially serious problem. Leo inserts an @ignore directive in the @auto tree if any error is found. This @ignore directive prevents the @auto tree from modifying the external file. If you @ignore directive, a later write of the @auto tree will attempt to fix the problems that gave rise to the errors. There are no guarantees however.

Strict languages are languages like Python for which leading whitespace is especially significant. Before importing a file for a strict language, Leo **regularizes** the leading whitespace of all lines of the original source file. That is, Leo converts blanks to tabs or tabs to blanks depending on the value of the @tabwidth directive in effect for the @auto node. Leo cannot guarantee to reproduce the original source file exactly if problems are discovered while regularizing leading whitespace.

After importing a file, Leo verifies that writing the @auto node would create the same file as the original file. For strict languages, the comparison must be exact, or nearly so. For non-strict languages, differences in leading whitespace generate warnings, not errors.

File comparison mismatches can arise for several reasons:

1. Bugs in the import parsers. Please report any suspected bugs immediately.
2. Underindented lines in classes, methods or functions in strict languages. An **underindented line** is a line that is indented less then the starting line of the class, method or function in which it appears. Leo outlines can not represent such lines exactly: every line of node implicitly has at least the indentation of any unindented line of the node.

Leo will issue a warning (not an error) for underindented Python comment lines. Such lines can not change the meaning of Python programs.

Commands related to @auto

Three new commands in the File:Read/Write menu allow you to manually read and write @auto nodes from the presently selected outline. As always, an @ignore directive in the @auto node or its ancestors will suppress any of these commands:

- The Read @auto Nodes (read-at-auto-nodes) command reads all @auto nodes in the presently selected outline. An @ignore directive will suppress this import.
- The Write @auto Nodes (write-at-auto-nodes) command writes all @auto nodes. An @ignore directive will suppress this import. Caution: the write will occur even if Leo has not previously read the @auto node.

- The Write Dirty @auto Nodes (write-dirty-at-auto-nodes) is the same as the write-at-auto-nodes command, except that only changed @auto trees are written.

Most users will rarely use these explicit commands, because reading and writing .leo files handles @auto nodes well enough. However, you can use the read-at-auto-nodes command to update @auto nodes without having to reload the .leo file.

Extending the code: adding new parsers

All present parsers are short overrides of a powerful base parser class. Thus, it would be simple to add support for other languages. See the node:

```
@file leoImport.py-->Import-->Scanners for createOutline
```

in leoPy.leo to see how easy it is to create new parsers.

New commands for resolving cvs conflicts

The so-called resolve-cvs-conflict project has resolved itself into small, easily understood commands.

The **read-file-into-node** command prompts for a filename, and creates an node whose headline is @read-file-into-node <filename> and whose body text is the entire contents of the file.

The **write-file-from-node** command writes the body text of the selected not to a file. If the headline of the presently selected node starts with @read-file-into-node the command use the filename that follows in the headline. Otherwise, the command prompts for a filename.

When a cvs conflict occurs, the user will:

- read the file into a node using the read-file-into-node command,
- fix the conflict, as with any other editor, and
- write the file with the write-file-from-node command.

Any file can be fixed in this way, including external files and .leo files. The only complication is that the user must not change sentinel lines. Two new commands check the contents of a node: The **check-derived-file** and **check-leo-file** commands tell whether a trial read of the presently selected node can be done successfully. The check-derived-file command assumes the body text is a external file; the check-leo-file command assumes the body text is an entire .leo file.

The **compare-leo-outlines** command prompts for another (presumably similar) .leo file that will be compared with the presently selected outline file (main window). It then creates clones of all inserted, deleted and changed nodes.

New kinds of settings trees

@buttons trees

All @buttons tree in a settings file defines global buttons that are created in the icon area of all .leo files. You define @button nodes in the @buttons tree as usual.

@menus trees

Leo creates its menus from the @menu and @item nodes in the @menus tree. Within @menus trees, @menu nodes create menus and @item nodes create menu items.

The menu name always follows @menu. If the menu name is `Plugins', Leo will create the Plugins menu and populate the menu by calling the `create-optional-menus' hook. This creates the Plugins menu as usual. Nested @menu nodes define submenus.

The command name follows @item. If the body text of an @item node exists, this body text is the menu name. Otherwise, the menu name is the command name. However, if the command name starts with a `*', hyphens are removed from the menu name. Menu names and command names may contain a single ampersand (&). If present, the following character is underlined in the name. If the command name in an @item node is just a hyphen (-), the item represents a menu separator.

New plugins

- The graphed plugin allows users to manipulate parts of Leo outlines as if they were general graphs. It is still early days for this exciting plugin.
- The threading_colorizer plugin replaces the __jEdit_colorizer__ plugin. This plugin features an elegant new algorithm that has much better performance and eliminates almost all flash.

Leo's core is now compatible with jython

Essentially all of Leo's startup code now runs with jython 2.2 and the (unfinished!) swing gui.

Improved prototype for icons in headlines

The prototype in test.leo now will use PIL (Python Imaging Library) if available, so many more kinds of icons can be used. Buttons now exist to add icons to do the following:

- Add any icon to any node.
- Delete all icons from a single node or the entire tree.
- Print the icon files associated with a node.
- Print the sizes of icons in a directory.

Fixed a bug in the icon handling in the outline widget that caused duplicate icons not to be drawn properly.

Minor improvements

- See the release notes for a list of bugs fixed in Leo 4.4.4.
- Added the `clear-all-marks' hook.
- Added button font setting. See the node:

```
"@settings-->Fonts-->@font button font" in leoSettings.leo.
```

- Plugins and scripts may call the c.frame.canvas.createCanvas method to create a log tab containing a Tk.Canvas widget. Here is an example script:

```
log = c.frame.log ; tag = 'my-canvas'
w = log.canvasDict.get(tag)
if not w:
    w = log.createCanvas(tag)
    w.configure(bg='yellow')
log.selectTab(tag)
```

- Improved the yank and yank-pop commands and added @bool add_ws_to_kill_ring setting.
- Improved the debug command: it now adds the following code to the beginning of debug scripts:

```
class G:
    def es(s,c=None):
      pass
g = G()
```

- Added the @bool rst3 strip_at_file_prefixes setting.
- Added the g.app.inBridge ivar.
- Added @bool big_outline_pane setting. False (legacy): Top pane contains outline and log panes. True: Top pane contains only the outline pane. Bottom pane contains body and log panes.

Summary of new commands

```
check-derived-file
check-leo-file
compare-leo-outlines
insert-child
read-at-auto-nodes
read-file-into-node
write-at-auto-nodes
write-dirty-at-auto-nodes
write-file-from-node
```

31.2.10 Leo 4.4.3

The highlights of Leo 4.4.3:

- @test and @suite nodes may now be embedded directly in external files.
- Added support for chapters in Leo's core.
- Added support for zipped .leo files.
- The new leoBridge module allows full access to all of Leo's capabilities from programs running outside of Leo.
- Better support for the winpdb debugger.
- Added support for @enabled-plugins and @open-with nodes in settings files.

- Removed all gui-dependent code from Leo's core.
- The__wx_gui plugin is now functional.

31.2.11 Leo 4.4.2

A major code reorg

Leo's vnode and tnode classes are now completely independent of the rest of Leo. Some api's have been changed. This `big reorg' and may affect scripts and plugins.

New commands

```
extend-to-line
extend-to-paragraph
extend-to-sentence
forward-end-word
forward-end-word-extend-selection
```

New features

- Added support for controlling Leo from Emacs with pymacs. See the Leo and Emacs chapter for full details.
- Added Minibuffer and Settings submenus of the Cmds menu.
- At long last Leo creates a proper help menu on the Mac.
- Added a new convention for menu tables. If the first item (a string representing the menu label) starts with `*' Leo will convert hyphens to spaces and upcase the label. This convention allows a single string to represent both the menu label and its associated minibuffer command. As part of this reorganization, all menu tables in Leo's core now use only strings. This is an essential precondition to supporting @menu nodes in leoSettings.leo.
- Leo's Help menu now contains the Open scripts.leo command.
- Leo uses ctypes to import Aspell when run from Python 2.5 or later. Leo no longer needs Python-specific versions of aspell.dll.
- Added support for x-windows middle-button paste. This only works when the paste is made in the pane containing the selected text.
- Leo looks for myLeoSettings.leo files in the same place Leo looks for leoSettings.leo files.
- Created three scripts (in test.leo) that help create unit tests for Leo's edit commands. Create Created runEditCommandTest for use by these scripts.
- Improved print-bindings command. The bindings are sorted by prefix: this is a big help in understanding bindings. For each prefix, first print items with only a single character after the prefix.
- Made writing .leo files faster. The new code almost exactly twice as fast as the old.
- Added p.archivedPosition. This is a key first step towards Leap 204.
- Integrated sax with read logic.

- You can now store settings in myLeoSettings.leo without fear of those settings being changed by cvs updates or in future versions of Leo.
- Eliminated unnecessary redraws when moving the cursor in the outline pane.
- Much faster navigation through the outline using Alt-arrow keys.
- When focus is in the outline pane, you can move to headlines by typing the first letter of headlines.
- The find command now closes nodes not needed to show the node containing the present match.
- Numerous changes that make Leo easier to use without using a mouse.
- Many new minibuffer commands now appear in the Cmds menu.

Further improved outline navigation:

- Generalized navigation in outline pane to ignore @file, @thin, etc prefixes.
- Made outline navigation cumulative. When keystrokes in the outline pane are typed `close' together Leo first tries to look for prefix + ch, where ch is the character just typed and prefix is the previous match. The term `close together' is specified by the setting @float outline_nav_extend_delay. The outline search revers to a single-character if the extended search fails, so in fact the delay is not too significant. In practice everything works well without me thinking at all about what is happening.

New and improved plugins

- Improved the mod_scripting plugin. Every button created by the plugin creates a corresponding command. The command name is the `cleaned' version of the button name. Likewise, the plugin also creates a delete-x-button command, where x is the command name as just discussed. So now you can delete script buttons without right-clicking.
- Made `About Plugin' dialog scrollable.
- Fixed bugs in groupoperations, multifile, nodenavigator and shortcut_button plugins.
- The rst3 plugin now registers the rst3-process-tree command.
- The leoOPML.py plugin defines commands to read and write OPML files.
- The slideshow.py plugin allows Leo to run slideshows defined by @slideshow and @slide nodes.
- The leo_to_rtf and leo_to_html plugins create rtf and html files from Leo outlines.
- The paste_as_headlines.py plugins creates multiple headlines at once.
- The word_count.py plugin.

Improved the mod_scripting plugin:

- Made showing the Run Script button optional.
- The Script Button button now creates the press-script-button-button command.
- A new utility method does a much better job of massaging button and command names.

Settings

- Removed .leoRecentFiles.txt from the distribution and cvs and added @bool write_recent_files_as_needed. The presence or absence of .leoRecentFiles.txt no longer controls whether Leo creates and updates .leoRecentFiles.txt.
- Added @bool insert_new_nodes_at_end.
- Added @bool select_all_text_when_editing_headlines. Creating a new node always selects the entire text, regardless of this option.
- Leo looks for myLeoSettings.leo files in the same place Leo looks for leoSettings.leo files.
- Added settings for all mod_scripting switches.
- Added @bool collapse_nodes_during_finds. This greatly speeds searches that used to open many nodes. See: http://sourceforge.net/forum/message.php?msg_id=3935780
- Added @bool outline_pane_has_initial_focus.
- Added @bool sparse_move_outline_left.
- Added bindings for Alt-Shift-Arrow keys to force an outline move.
- Added @bool use_sax_based_read = False. True: Use a sax-based parser to read .leo files. This is slower than using Leo's legacy xml parser, but may solve some unicode problems.

Changed default settings:

```
focus-to-body = Alt-D
focus-to-tree = Alt-T
toggle-extend-mode = Alt-3
```

ZODB scripting

Leo's vnode and tnode classes can optionally be compatible with ZODB databases, i.e., they can optionally derive from ZODB.Persistence.Persistent. See Chapter 17: Using ZODB with Leo for details.

31.2.12 Leo 4.4.1

The main features of Leo 4.4.1 are:

- Multiple editors in Leo's body pane and
- A new colorizer plugin controlled by jEdit language description files.
- Search commands now support regex replace patterns: 1, 2, etc.
- Support for external debuggers: see http://webpages.charter.net/edreamleo/debuggers.html
- The scripting plugin now creates a Debug Script button.
- Several new commands including run-unit-test, python-help and toggle-invisibles.
- The help-for-command commands now contains information for almost all commands.
- A new shortcut_button plugin.

New commands

```
cycle-focus
debug
find-character
find-word
hide-invisibles
isearch-with-present-options
open-users-guide
python-help
run-unit-test
toggle-autocompleter
toggle-calltips
toggle-invisibles
```

New features

- Removed warning about changed node.
- Added scroll-outline-left/right commands.
- Leo outputs decorators correctly, assuming the decorator does not conflict with a Leo directive.
- Wrote script to convert g.es to g.et where appropriate. The first step in translating all Leo messages.
- Leo highlights (flashes) matching brackets when typing typing (,), [,], { or }.
- Fixed long-standing problem reporting indentation errors.
- Fixed long-standing bug in Remove Sentinels command.
- Fixed long-standing bugs in import commands.
- The scroll-up/down commands now scroll the outline if focus is in outline pane. However, his can be done better using per-pane bindings as in the default leoSettings.leo.
- Incremental searches are (properly) confined to a single body text.
- Backspace now handled properly in incremental searches.
- The add-editor command adds a new editor in the body pane. The delete-editor command deletes the presently selected editor, and the cycle-editor-focus command cycles focus between editors in the body text.
- The standard 1, 2, etc. replacements can now be performed in regular expression searches.
- The standard escapes n and t are now valid in plain searches.
- The shortcut for the replace-string command now changes from the find command to the replace command.

New and improved plugins

- The slideshow plugin
- The mod_scripting plugin now creates a press-x-button command for every button `x'. You can specify settings for such commands using @shortcuts nodes.

- The shortcut_button plugin plugin creates a `Shortcut' button in the icon area. Pressing the Shortcut button creates *another* button which when pressed will select the presently selected node at the time the button was created.
- Added Debug button to scripting plugin.

New settings

```
@bool autoindent_in_nocolor_mode
@bool flash_matching_brackets
@bool idle_redraw
@bool trace_bind_key_exceptions
@bool warn_about_redefined_shortcuts
@color flash_brackets_background_color
@color flash_brackets_foreground_color
@int flash-brackets-delay
@int flash_brackets_count
@string close_flash_brackets
@string open_flash_brackets
@string editor_orientation
```

Improved settings

- Added @font menu font setting.
- Added support for commands to be executed on entry to a mode.
- Added support for bindings that are active only in command, enter and insert key states.
- Added support for @abbrev nodes in leoSettings.leo.
- Improved check bindings script in leoSettings.leo.
- Allow @mode outside of leoSettings.leo.
- Added warnings about the @bool expanded_click_area setting.

Minor improvements

- The print-bindings command now properly sorts bindings.
- The help-for-command command now works for almost all commands.
- Improved filename completion.
- Better listings for print-commands and print-bindings &amp; mode-help commands.
- Allow shortcuts to be overridden outside of leoSettings.leo.
- Finished Cmds menu.
- Improved show-fonts command.
- Strip quotes from color, font settings.

- Warn about invalid Enter and Leave key bindings.

31.2.13 Leo 4.4

The main features of Leo 4.4 are:

- An Emacs-like mini-buffer: you can now execute any command by typing its long name, with tab completion.
- Many new commands, including cursor and screen movement, basic character, word and paragraph manipulation, and commands to manipulate buffers, the kill ring, regions and rectangles. You can use Leo without using a mouse.
- Flexible key bindings and input modes. You can emulate the operation of Emacs, Vim, or any other editor.
- A tabbed log pane. The Find and Spell Check commands now use tabs instead of dialogs, making those commands much easier to use. Plugins or scripts can easily create new tabs. The Completion tab shows possible typing completions.
- Autocompletion and calltips. Autocompletion works much like tab completion. To enable autocompletion, bind a key to the auto-complete command.

New commands

```
activate-cmds-menu
activate-edit-menu
activate-file-menu
activate-help-menu
activate-outline-menu
activate-plugins-menu
activate-window-menu
add-space-to-lines
add-tab-to-lines
clean-lines
clear-selected-text
click-click-box
click-headline
click-icon-box
clone-find-all
contract-and-go-right
contract-body-pane
contract-log-pane
contract-outline-pane
contract-pane
double-click-headline
double-click-icon-box
dump-all-objects
dump-new-objects
expand-body-pane
expand-log-pane
expand-outline-pane
expand-pane
```

```
find-again
find-all
find-tab-change
find-tab-change-all
find-tab-change-then-find
find-tab-find command
find-tab-find-previous
free-text-widgets
fully-expand-body-pane
fully-expand-log-pane
fully-expand-outline-pane
fully-expand-pane
goto-first-sibling
goto-global-line
goto-last-sibling
help
help-for-command
hide-body-pane
hide-find-tab
hide-log-pane
hide-minibuffer
hide-outline-pane
hide-pane,
open-find-tab
open-find-tab
open-outline-by-name (uses filename completion)
open-spell-tab
print-bindings
print-commands    re-search-backward
re-search-forward
remove-space-from-lines
remove-tab-from-lines
replace-string
scroll-down
scroll-down-extend-selection
scroll-outline-down-line
scroll-outline-down-page
scroll-outline-up-line
scroll-outline-up-page
scroll-up
scroll-up-extend-selection
search-backward
search-forward
search-with-present-options
set-find-everywhere
set-find-node-only
set-find-suboutline-only
show-colors
```

```
show-fonts
show-minibuffer
show-search-options
simulate-begin-drag
simulate-end-drag
toggle-find-ignore-case-option
toggle-find-in-body-option,
toggle-find-in-headline-option
toggle-find-mark-changes-option
toggle-find-mark-finds-option
toggle-find-regex-option
toggle-find-reverse-option
toggle-find-word-option and
toggle-find-wrap-around-option
toggle-mini-buffer
verbose-dump-objects
word-search-backward
word-search-forward
```

New features

- Added script to update new copies of leoSetttings.leo from previous copies.
- Made all edit command undoable.
- Improved registerCommand.
- Suppressed autocompletion after numbers.
- Added colorizing support for Lua language.
- Added run-unit-test command.
- Autocompletion and calltips.
- Leo remembers the previous open directory.
- Fixed problem with view plugin.
- Installed cleo patch.
- User input modes.
- Installed many standard bindings to leoSettings.leo.
- Added Check Bindings script in leoSettings.leo.
- Scripts now maintain original focus.
- Improved cursor move/extend commands.
- Added support for @mode nodes.
- keyboard-quit restores default input mode.
- Created ut.leo, ut.py and ut.bat.

- Added modes/*.xml to distribution.
- Revised cursor movement commands and added selection-extension commands.
- Added classic key bindings in leoSettings.leo.
- Allow multiple key bindings to the same command.
- Settings command now opens leoSettings.leo.
- Moved all scripts into scripts.leo.
- Improved how the New Tab and Rename Tab commands work in the log pane.
- Improved the appearance of the Spell tab.
- Added Clone-find checkbox to the Find tab.
- Improved find tab.
- Improved formatting of shortcuts in print-commands and print-bindings.
- Added settings for vim plugin.
- Put up a dialog if can't import Pmw.
- Bound <Return> to end-edit-headline.
- Leo now ignores key bindings in menu tables.
- Created scripts.leo and unitTest.leo.
- c.executeMinibufferCommand executes a minibuffer command by name.
- Improved perl entries in language dicts.
- The tabbed log.
- The Find tab replaces the old Find panel; the old Find panel is deprecated.

New and improved plugins

- Changed path to stylesheet in the rst3 plugin.
- Fixed crasher in Word (and other) plugins.
- Fixed problem with labels plugin.
- Added the following commands for the groupoperations plugin:

```
group-operations-clear-marked
group-operations-mark-for-copy
group-operations-mark-for-move
group-operations-mark-for-clone
group-operations-mark-target
group-operations-operate-on-marked
group-operations-transfer
```

- Installed cleo patch.

- The scripting plugin now supports shortcuts in @button nodes:

```
@button name @key=shortcut
```

- The scripting plugin now supports @command nodes:

```
@command name @key=shortcut
```

New and improved settings

Added new settings:

```
@bool allow_idle_time_hook
@bool autocomplete-brackets.
@bool gc_before_redraw
@bool minibufferSearchesShowFindTab
@bool show_only_find_tab_options
@bool show_tree_stats
@bool trace_autocompleter
@bool trace_bindings
@bool trace_doCommand
@bool trace_f.set_focus
@bool trace_focus
@bool trace_g.app.gui.set_focus
@bool trace_gc
@bool trace_gc_calls
@bool trace_gc_verbose
@bool trace_key_event
@bool trace_masterClickHandler
@bool trace_masterCommand
@bool trace_masterFocusHandler
@bool trace_masterKeyHandler
@bool trace_minibuffer
@bool trace_modes
@bool trace_redraw_now
@bool trace_select
@bool trace_status_line
@bool trace_tree
@bool trace_tree_alloc
@bool trace_tree_edit
@bool useCmdMenu
@bool useMinibuffer
@bool use_syntax_coloring
@color body_text_selection_background_color
@color body_text_selection_foreground_color.
@color log_pane_Find_tab_background_color
@color log_pane_Spell_tab_background_color, etc.
@int max_undo_stack_size,
@string trace_bindings_filter
```

```
@string trace_bindings_pane_filter
```

- Added @shortcuts nodes.
- Leo now supports per-pane bindings of the form:

```
    command-name ! pane = shortcut
```

- The spelling settings replace the settings in spellpyx.ini.

Leo's home page