Leo API Documentation

Release 4.11dev

Ed K. Ream

May 31, 2018

Contents

Contents:

CHAPTER 1

leo Package

1.1 leo Package

leo.__init__.**run**(**args*, ***keys*)

1.2 Subpackages

1.2.1 core Package

1.2.1.1 core Package

1.2.1.2 bzr_version Module

1.2.1.3 format-code Module

1.2.1.4 leoApp Module

class leo.core.leoApp.**IdleTimeManager**
Bases: object

A singleton class to manage idle-time handling. This class handles all details of running code at idle time, including running 'idle' hooks.

Any code can call g.app.idleTimeManager.add_callback(callback) to cause the callback to be called at idle time forever.

add_callback(*callback*)
Add a callback to be called at every idle time.

on_idle(*timer*)
IdleTimeManager: Run all idle-time callbacks.

on_idle_count = 0

start()
Start the idle-time timer.

class leo.core.leoApp.**LeoApp**
Bases: object

A class representing the Leo application itself.

Ivars of this class are Leo's global variables.

checkForOpenFile(*c*, *fn*)
Warn if fn is already open and add fn to already_open_files list.

closeLeoWindow(*frame*, *new_c=None*, *finish_quit=True*)
Attempt to close a Leo window.

Return False if the user veto's the close.

finish_quit - usually True, close Leo when last file closes, but False when closing an already-open-elsewhere file during initial load, so UI remains for files further along the command line.

cmd()
Command decorator for the LeoApp class.

commanders()
Return list of currently active controllers

computeSignon()

createCursesGui(*fileName=''*, *verbose=False*)

createDefaultGui(*fileName=''*, *verbose=False*)
A convenience routines for plugins to create the default gui class.

createNullGuiWithScript(*script=None*)

createQtGui(*fileName=''*, *verbose=False*)
A convenience routines for plugins to create the Qt gui class.

createTextGui(*fileName=''*, *verbose=False*)

createWxGui(*fileName=''*, *verbose=False*)
A convenience routines for plugins to create the wx gui class.

define_delegate_language_dict()

define_extension_dict()

define_global_constants()

define_language_delims_dict()

define_language_extension_dict()

destroyAllOpenWithFiles()
Remove temp files created with the Open With command.

destroyWindow(*frame*)
Destroy all ivars in a Leo frame.

finishQuit()

forceShutdown()
Forces an immediate shutdown of Leo at any time.

In particular, may be called from plugins during startup.

forgetOpenFile(*fn*, *force=False*)
Forget the open file, so that is no longer considered open.

init_at_auto_names()
Init the app.atAutoNames set.

init_at_file_names()
Init the app.atFileNames set.

listenToLog(*event=None*)
A socket listener, listening on localhost. See: https://docs.python.org/2/howto/logging-cookbook.html#sending-and-receiving-logging-events-across-a-network

Start this listener first, then start the broadcaster.

leo/plugins/cursesGui2.py is a typical broadcaster.

lockLog()
Disable changes to the log

makeAllBindings()
LeoApp.makeAllBindings:

Call c.k.makeAllBindings for all open commanders c.

newCommander(*fileName*, *relativeFileName=None*, *gui=None*, *previousSettings=None*)
Create a commander and its view frame for the Leo main window.

onQuit(*event=None*)
Exit Leo, prompting to save unsaved outlines first.

rememberOpenFile(*fn*)

runAlreadyOpenDialog(*c*)
Warn about possibly already-open files.

scanner_for_at_auto(*c*, *p*, ***kwargs*)
A factory returning a scanner function for p, an @auto node.

scanner_for_ext(*c*, *ext*, ***kwargs*)
A factory returning a scanner function for the given file extension.

selectLeoWindow(*c*)

setGlobalDb()
Create global pickleshare db

Usable by:

```
g.app.db['hello'] = [1,2,5]
```

setIDFile()
Create leoID.txt.

setIDFromEnv(*verbose*)
Set leoID from environment vars.

setIDFromFile(*verbose*)
Attempt to set g.app.leoID from leoID.txt.

setIDFromSys(*verbose*)
Attempt to set g.app.leoID from sys.leoID.

This might be set by in Python's sitecustomize.py file.

setIdFromDialog()
Get leoID from a dialog.

setLeoID(*useDialog=True*, *verbose=True*)
Get g.app.leoID from various sources.

setLog(*log*)
set the frame to which log messages will go

unlockLog()
Enable changes to the log

writeWaitingLog(*c*)
Write all waiting lines to the log.

class leo.core.leoApp.**LoadManager**
Bases: object

A class to manage loading .leo files, including configuration files.

addOptionsToParser(*parser*)

adjustSysPath()
Adjust sys.path to enable imports as usual with Leo.

This method is no longer needed:

1. g.importModule will import from the 'external' or 'extensions' folders as needed without altering sys.path.
2. Plugins now do fully qualified imports.

checkForDuplicateShortcuts(*c*, *d*)
Check for duplicates in an "inverted" dictionary d whose keys are strokes and whose values are lists of BindingInfo nodes.

Duplicates happen only if panes conflict.

completeFileName(*fileName*)

computeBindingLetter(*kind*)

computeFilesList(*options*, *fileName*)
Return the list of files on the command line.

computeGlobalConfigDir()

computeHomeDir()
Returns the user's home directory.

computeHomeLeoDir()

computeLeoDir()

computeLeoSettingsPath()
Return the full path to leoSettings.leo.

computeLoadDir()
Returns the directory containing leo.py.

computeLocalSettings(*c*, *settings_d*, *bindings_d*, *localFlag*)
Merge the settings dicts from c's outline into *new copies of* settings_d and bindings_d.

computeMachineName()
Return the name of the current machine, i.e, HOSTNAME.

computeMyLeoSettingsPath()
Return the full path to myLeoSettings.leo.

The "footnote": Get the local directory from lm.files[0]

computeStandardDirectories()
Compute the locations of standard directories and set the corresponding ivars.

computeThemeDirectories()
Return a list of *existing* directories that might contain theme .leo files.

computeThemeFilePath()
Return the absolute path to the theme .leo file.

computeWorkbookFileName()
Return the name of the workbook.

Return None *only* if: 1. The workbook does not exist. 2. We are unit testing or in batch mode.

createAllImporetersData()
New in Leo 5.5:

Create global data structures describing importers and writers.

createDefaultSettingsDicts()
Create lm.globalSettingsDict & lm.globalBindingsDict.

createGui(*pymacs*)

createImporterData()
Create the data structures describing importer plugins.

createMenu(*c*, *fn=None*)

createSettingsDicts(*c*, *localFlag*, *theme=False*)

createSpecialGui(*gui*, *pymacs*, *script*, *windowFlag*)

createWritersData()
Create the data structures describing writer plugins.

doDiff()
Support –diff option after loading Leo.

doGuiOption(*options*)

doLoadTypeOption(*options*)

doPostPluginsInit()
Create a Leo window for each file in the lm.files list.

doPrePluginsInit(*fileName*, *pymacs*)
Scan options, set directories and read settings.

doScreenShotOption(*options*)

doScriptOption(*options*, *parser*)

doSimpleOptions(*options*)
These args just set g.app ivars.

doWindowSizeOption(*options*)

findOpenFile(*fn*)

finishOpen(*c*)

getDefaultFile()

getPreviousSettings(*fn*)
Return the settings in effect for fn. Typically, this involves pre-reading fn.

initApp(*verbose*)

initFocusAndDraw(*c*, *fileName*)

initWrapperLeoFile(*c*, *fn*)
Create an empty file if the external fn is empty.

Otherwise, create an @edit or @file node for the external file.

invert(*d*)
Invert a shortcut dict whose keys are command names, returning a dict whose keys are strokes.

isLeoFile(*fn*)

isValidPython()

isZippedFile(*fn*)

load(*fileName=None*, *pymacs=None*)
Load the indicated file

loadLocalFile(*fn*, *gui*, *old_c*)
Completely read a file, creating the corresonding outline.

1. If fn is an existing .leo file (possibly zipped), read it twice: the first time with a NullGui to discover settings, the second time with the requested gui to create the outline.

2. If fn is an external file: get settings from the leoSettings.leo and myLeoSetting.leo, then create a "wrapper" outline continain an @file node for the external file.

3. If fn is empty: get settings from the leoSettings.leo and myLeoSetting.leo or default settings, or open an empty outline.

make_screen_shot(*fn*)
Create a screenshot of the present Leo outline and save it to path.

mergeShortcutsDicts(*c*, *old_d*, *new_d*, *localFlag*)
Create a new dict by overriding all shortcuts in old_d by shortcuts in new_d.

Both old_d and new_d remain unchanged.

openEmptyWorkBook()
Open an empty frame and paste the contents of CheatSheet.leo into it.

openFileByName(*fn*, *gui*, *old_c*, *previousSettings*)
Read the local file whose full path is fn using the given gui. fn may be a Leo file (including .leo or zipped file) or an external file.

This is not a pre-read: the previousSettings always exist and the commander created here persists until the user closes the outline.

Reads the entire outline if fn exists and is a .leo file or zipped file. Creates an empty outline if fn is a non-existent Leo file. Creates an wrapper outline if fn is an external file, existing or not.

openLeoFile(*fn*)

openLeoOrZipFile(*fn*)

openSettingsFile(*fn*)
Open a settings file with a null gui. Return the commander.

The caller must init the c.config object.

openZipFile(*fn*)

parse_importer_dict(*sfn*, *m*)
Set entries in g.app.classDispatchDict, g.app.atAutoDict and g.app.atAutoNames using entries in m.importer_dict.

parse_writer_dict(*sfn*, *m*)
Set entries in g.app.writersDispatchDict and g.app.atAutoWritersDict using entries in m.writers_dict.

readGlobalSettingsFiles()
Read leoSettings.leo and myLeoSettings.leo using a null gui.

readOpenedLeoFile(*c*, *fn*, *readAtFileNodesFlag*, *theFile*)

reportDirectories(*verbose*)
Report directories.

resolve_theme_path(*fn*, *tag*)
Search theme directories for the given .leo file.

revertCommander(*c*)
Revert c to the previously saved contents.

scanOptions(*fileName*, *pymacs*)
Handle all options, remove them from sys.argv and set lm.options.

setStdStreams()
Make sure that stdout and stderr exist. This is an issue when running Leo with pythonw.exe.

traceSettingsDict(*d*, *verbose=False*)

traceShortcutsDict(*d*, *verbose=False*)

uninvert(*d*)
Uninvert an inverted shortcut dict whose keys are strokes, returning a dict whose keys are command names.

class leo.core.leoApp.**PreviousSettings**(*settingsDict*, *shortcutsDict*)
Bases: object

A class holding the settings and shortcuts dictionaries that are computed in the first pass when loading local files and passed to the second pass.

class leo.core.leoApp.**RecentFilesManager**
Bases: object

A class to manipulate leoRecentFiles.txt.

appendToRecentFiles(*files*)

cleanRecentFiles(*c*)
Remove items from the recent files list that no longer exist.

This almost never does anything because Leo's startup logic removes nonexistent files from the recent files list.

clearRecentFiles(*c*)
Clear the recent files list, then add the present file.

createRecentFiles()
Try to create .leoRecentFiles.txt, in the users home directory, or in Leo's config directory if that fails.

createRecentFilesMenuItems(*c*)

demangleRecentFiles(*c*, *data*)
Rewrite recent files based on c.config.getData('path-demangle')

editRecentFiles(*c*)
Dump recentFiles into new node appended as lastTopLevel, selects it and request focus in body.

NOTE: command write-edited-recent-files assume that headline of this node is not changed by user.

getRecentFiles()

getRecentFilesTable()

readRecentFiles(*localConfigFile*)
Read all .leoRecentFiles.txt files.

readRecentFilesFile(*path*)

sanitize(*name*)
Return a sanitized file name.

setRecentFiles(*files*)
Update the recent files list.

sortRecentFiles(*c*)
Sort the recent files list.

updateRecentFiles(*fileName*)
Create the RecentFiles menu. May be called with Null fileName.

writeEditedRecentFiles(*c*)
Write content of "edit_headline" node as recentFiles and recreates menues.

writeRecentFilesFile(*c*, *force=False*)
Write the appropriate .leoRecentFiles.txt file.

Write a message if force is True, or if it hasn't been written yet.

writeRecentFilesFileHelper(*fileName*)

leo.core.leoApp.**ctrlClickAtCursor**(*event*)
Simulate a control-click at the cursor.

leo.core.leoApp.**demangle_recent_files_command**(*event*)
Path demangling potentially alters the paths in the recent files list according to find/replace patterns in the @data path-demangle setting. For example:

REPLACE: .gnome-desktop WITH: My Desktop

The default setting specifies no patterns.

leo.core.leoApp.**disable_idle_time_events**(*event*)
Disable default idle-time event handling.

leo.core.leoApp.**enable_idle_time_events**(*event*)
Enable default idle-time event handling.

leo.core.leoApp.**join_leo_irc**(*event=None*)
Open the web page to Leo's irc channel on freenode.net.

leo.core.leoApp.**openUrl**(*event=None*)
Open the url in the headline or body text of the selected node.

Use the headline if it contains a valid url. Otherwise, look *only* at the first line of the body.

leo.core.leoApp.**openUrlUnderCursor**(*event=None*)
Open the url under the cursor.

leo.core.leoApp.**toggle_idle_time_events**(*event*)
Toggle default idle-time event handling.

1.2.1.5 leoAtFile Module

Classes to read and write @file nodes.

class leo.core.leoAtFile.**AtFile**(*c*)
Bases: object

A class implementing the atFile subcommander.

adjustTargetLanguage(*fn*)
Use the language implied by fn's extension if there is a conflict between it and c.target_language.

allDirective = 2

appendToDocPart(*s*)
Append the next line of the @doc part to docOut.

appendToOut(*s*)
Append s to at.out (old sentinels) or at.v.tempBodyList (new sentinels).

asisWrite(*root*, *toString=False*)

atDirective = 4

at_directive_kind_pattern = <_sre.SRE_Pattern object>

autoBeautify(*p*)
Auto beautify p's tree if allowed by settings and directives.

badEndSentinel(*expectedKind*)
Handle a mismatched ending sentinel.

bodyIsInited(*v*)
Return True if v.b has been inited.

bodySetInited(*v*)
Indicate that v.b has been inited.

cDirective = 6

changeLevel(*oldLevel*, *newLevel*)
Update data structures when changing node level.

The key invariant: on exit, the top of at.thinNodeStack is the new parent node.

checkDerivedFile(*event=None*)
Make sure an external file written by Leo may be read properly.

checkExternalFileAgainstDb(*root*)
Returns True if file is not modified since last save in db. Otherwise returns False.

checkPythonCode(*root*, *s=None*, *targetFn=None*, *pyflakes_errors_only=False*)
Perform python-related checks on root.

checkPythonSyntax(*p*, *body*, *supress=False*)

chmod(*fileName*, *mode*)

clearAllBodyInited()
Clear all v.b inited bits.

clearAllOrphanBits(*p*)
Clear orphan bits for all nodes *except* orphan @file nodes.

closeStringFile(*theFile*)

closeWriteFile()

cmd()
Command decorator for the AtFileCommands class.

codeDirective = 5

compareFiles(*path1*, *path2*, *ignoreLineEndings*, *ignoreBlankLines=False*)
Compare two text files.

completeFirstDirectives(*out*, *firstLines*)
14-SEP-2002 DTHEIN

Scans the lines in the list 'out' for @first directives, appending the corresponding line from 'firstLines' to each @first directive found.

NOTE: the @first directives must be the very first lines in 'out'.

completeLastDirectives(*out*, *lastLines*)
14-SEP-2002 DTHEIN.

Scans the lines in the list 'out' for @last directives, appending the corresponding line from 'lastLines' to each @last directive found.

NOTE: the @last directives must be the very last lines in 'out'.

completeRootNode(*firstLines*, *lastLines*, *root*)
Terminate the root's body text, handling @first and @last.

create(*fn*, *s*)
Create a file whose contents are s.

createImportedNode(*root*, *headline*)

createNewThinNode(*gnx*, *headline*, *level*)
Create a new (new-style) vnode.

createResurrectedNodesNode()
Create a 'Resurrected Nodes' node as the last top-level node.

createV5ThinNode(*gnx*, *headline*, *level*)
Create a version 5 vnode.

defineDispatchDict()
Return the dispatch dictionary used by scanText4.

defineResurrectedNodeCallback(*r*, *root*)
Define a callback that moves node p as r's last child.

deleteAllTempBodyStrings()
Delete all temp attributes.

deleteTnodeList(*p*)
Remove p's tnodeList.

deleteUnvisitedNodes(*root*, *redraw=True*)
Delete unvisited nodes in root's subtree, not including root.

Before Leo 5.6: Move unvisited node to be children of the 'Resurrected Nodes'.

directiveKind4 (*s*, *i*)
Return the kind of at-directive or noDirective.

dispatch (*ext*, *p*)
Return the correct writer function for p, an @auto node.

docDirective = 3

dump (*lines*, *tag*)
Dump all lines.

endAll = 70

endAt = 21

endBody = 22

endDoc = 24

endLeo = 25

endMiddle = 71

endNode = 26

endOthers = 27

endRawDirective = 10

endRef = 84

ensureTrailingNewline (*s*)
Ensure a trailing newline in s. If we add a trailing newline, we'll generate an @nonl sentinel below.

- We always ensure a newline in @file and @thin trees.
- This code is not used used in @asis trees.
- New in Leo 4.4.3 b1: We add a newline in @clean/@nosent trees unless @bool force_newlines_in_at_nosent_bodies = False

error (**args*)

exception (*message*)

findChild4 (*headline*)
Return the next VNode in at.root.tnodeList. Called only for **legacy** @file nodes.

tnodeLists are used *only* when reading @file (not @thin) nodes. tnodeLists compensate for not having gnx's in derived files!

findReference (*name*, *p*)
Find a reference to name. Raise an error if not found.

findSectionName (*s*, *i*)
Return n1, n2 representing a section name. The section name, *including* brackes is s[n1:n2]

getEncodingFromHeader (*fn*, *s*)
Return the encoding given in the @+leo sentinel, if the sentinel is present, or the previous value of at.encoding otherwise.

getNodeHeadline (*s*, *i*)
Set headline to the rest of the line. Don't strip leading whitespace.

getPathUa (*p*)

handleChangedNode(*new_body*, *old_body*, *p*, *thinFile*)
Set ancestor files dirty and support mod_labels plugin.

ignoreOldSentinel(*s*, *unused_i*)
Ignore an 3.x sentinel.

importAtShadowNode(*fn*, *p*)

indicateNodeChanged(*old*, *new*, *postPass*, *v*)
Add an entry to c.nodeConflictList. Called only from at.terminateBody.

initCommonIvars()
Init ivars common to both reading and writing.

The defaults set here may be changed later.

initFileName(*fromString*, *importFileName*, *root*)
Return the fileName to be used in messages.

initReadIvars(*root*, *fileName*, *importFileName=None*, *perfectImportRoot=None*, *atShadow=False*)

initReadLine(*s*)
Init the ivars so that at.readLine will read all of s.

initScanText4(*p*)
Init all ivars for at.scanText4().

initWriteIvars(*root*, *targetFileName*, *atEdit=False*, *atShadow=False*, *forcePythonSentinels=False*, *nosentinels=False*, *toString=False*)

isFileLike(*s*)
Return True if s has file-like sentinels.

isSectionName(*s*, *i*)

isSignificantTree(*p*)
Return True if p's tree has a significant amount of information.

massageAtDocPart(*s*)
Compute the final @doc part when block comments are used.

miscDirective = 8

new_createThinChild4(*gnxString*, *headline*, *n*, *parent*)
Find or create a new *vnode* whose parent (also a vnode) is at.lastThinNode. This is called only for @thin trees.

noDirective = 1

noSentinel = 20

nodeSentinelText(*p*)
Return the text of a @+node or @-node sentinel for p.

oblank()

oblanks(*n*)

old_createThinChild4(*gnxString*, *headline*)
Find or create a new *vnode* whose parent (also a vnode) is at.lastThinNode. This is called only for @thin trees.

onl()
Write a newline to the output stream.

onl_sent()
Write a newline to the output stream, provided we are outputting sentinels.

openAtShadowFileForReading(*fn*)
Open an @shadow for reading and return shadow_fn.

openFileForReading(*fromString=False*)
Open the file given by at.root. This will be the private file for @shadow nodes.

openFileForWriting(*root*, *fileName*, *toString*)

openFileForWritingHelper(*fileName*)
Open the file and return True if all went well.

openFileHelper(*fn*)
Open a file, reporting all exceptions.

openForWrite(*filename*, *wb='wb'*)
Open a file for writes, handling shadow files.

openStringFile(*fn*, *encoding='utf-8'*)

os(*s*)
Write a string to the output stream.

All output produced by leoAtFile module goes here.

otabs(*n*)

othersDirective = 7

outputStringWithLineEndings(*s*)

parseLeoSentinel(*s*)
Parse the sentinel line s. If the sentinel is valid, set at.encoding, at.readVersion, at.readVersion5.

parseNodeSentinel(*s*, *i*, *middle*)

parseThinNodeSentinel(*s*, *i*)

parseUnderindentTag(*s*)

popSentinelStack(*expectedKind*)
Pop an entry from endSentinelStack and check it.

printError(**args*)
Print an error message that may contain non-ascii characters.

promptForDangerousWrite(*fileName*, *kind*, *message=None*)
Raise a dialog asking the user whether to overwrite an existing file.

putAfterLastRef(*s*, *start*, *delta*)
Handle whatever follows the last ref of a line.

putAfterMiddleRef(*s*, *delta*)
Handle whatever follows a ref that is not the last ref of a line.

putAtAllBody(*p*)
Generate the body enclosed in sentinel lines.

putAtAllChild(*p*)

putAtAllLine(*s*, *i*, *p*)
Put the expansion of @all.

putAtFirstLines(*s*)
Write any @firstlines from string s. These lines are converted to @verbatim lines, so the read logic simply ignores lines preceding the @+leo sentinel.

putAtLastLines(*s*)
Write any @last lines from string s. These lines are converted to @verbatim lines, so the read logic simply ignores lines following the @-leo sentinel.

putAtOthersChild(*p*)

putAtOthersLine(*s*, *i*, *p*)
Put the expansion of @others.

putBlankDocLine()

putBody(*p*, *fromString=''*)
Generate the body enclosed in sentinel lines. Return True if the body contains an @others line.

putBuffered(*s*)
Put s, converting all tabs to blanks as necessary.

putCloseNodeSentinel(*p*)
End a node.

putCodeLine(*s*, *i*)
Put a normal code line.

putDelims(*directive*, *s*, *k*)
Put an @delims directive.

putDirective(*s*, *i*)
Output a sentinel a directive or reference s.

It is important for PHP and other situations that @first and @last directives get translated to verbatim lines that do *not* include what follows the @first & @last directives.

putDocLine(*s*, *i*)
Handle one line of a doc part.

Output complete lines and split long lines and queue pending lines. Inserted newlines are always preceded by whitespace.

putEndDocLine()
Write the conclusion of a doc part.

putIndent(*n*, *s=''*)
Put tabs and spaces corresponding to n spaces, assuming that we are at the start of a line.

Remove extra blanks if the line starts with the underindentEscapeString

putInitialComment()

putLeadInSentinel(*s*, *i*, *j*, *delta*)
Set at.leadingWs as needed for @+others and @+<< sentinels.

i points at the start of a line. j points at @others or a section reference. delta is the change in at.indent that is about to happen and hasn't happened yet.

putLine(*i*, *kind*, *p*, *s*, *status*)
Put the line at s[i:] of the given kind, updating the status.

putOpenLeoSentinel(*s*)
Write @+leo sentinel.

putOpenNodeSentinel(*p*, *inAtAll=False*)
Write @+node sentinel for p.

putPending(*split*)
Write the pending part of a doc part.

We retain trailing whitespace iff the split flag is True.

putRefAt(*name*, *ref*, *delta*)

putRefLine(*s*, *i*, *n1*, *n2*, *name*, *p*)
Put a line containing one or more references.

putSentinel(*s*)
Write a sentinel whose text is s, applying the CWEB hack if needed.

putStartDocLine(*s*, *i*, *kind*)
Write the start of a doc part.

rawDirective = 9

read(*root*, *importFileName=None*, *fromString=None*, *atShadow=False*, *force=False*)
Read an @thin or @file tree.

readAfterRef(*s*, *i*)
Read an @afterref sentinel.

readAll(*root*, *force=False*)
Scan positions, looking for @<file> nodes to read.

readAtShadowNodes(*p*)
Read all @shadow nodes in the p's tree.

readClone(*s*, *i*)

readComment(*s*, *i*)
Read an @comment sentinel.

readDelims(*s*, *i*)
Read an @delims sentinel.

readDirective(*s*, *i*)
Read an @@sentinel.

readEndAll(*unused_s*, *unused_i*)
Read an @-all sentinel.

readEndAt(*unused_s*, *unused_i*)
Read an @-at sentinel.

readEndDoc(*unused_s*, *unused_i*)
Read an @-doc sentinel.

readEndLeo(*unused_s*, *unused_i*)
Read an @-leo sentinel.

readEndMiddle(*s*, *i*)
Read an @-middle sentinel.

readEndNode(*unused_s*, *unused_i*, *middle=False*)
Handle old-style @-node sentinels. In the new scheme, only the post-pass terminates nodes.

readEndOthers(*unused_s*, *unused_i*)
Read an @-others sentinel.

readEndRef (*unused_s*, *unused_i*)
Read an @-<< sentinel.

readError (*message*)

readFileToUnicode (*fn*)
Carefully sets at.encoding, then uses at.encoding to convert the file to a unicode string. Calls at.initReadLine if all went well.

Sets at.encoding as follows: 1. Use the BOM, if present. This unambiguously determines the encoding. 2. Use the -encoding= field in the @+leo header, if present and valid. 3. Otherwise, uses existing value of at.encoding, which comes from:

1. An @encoding directive, found by at.scanAllDirectives.
2. The value of c.config.default_derived_file_encoding.

Returns the string, or None on failure.

This method is now part of the main @file read code. at.openFileForReading calls this method to read all @file nodes. Previously only at.scanHeaderForThin (import code) called this method.

readLastDocLine (*tag*)
Read the @c line that terminates the doc part. tag is @doc or @.

Not used when reading new sentinels.

readLine ()
Read one line from file using the present encoding. Returns at.read_lines[at.read_i++]

readNl (*s*, *i*)
Handle an @nonl sentinel.

readNonl (*s*, *i*)
Handle an @nonl sentinel.

readNormalLine (*s*, *i=0*)

readOneAtAutoNode (*fileName*, *p*)
Read an @auto file into p. Return the *new* position.

readOneAtCleanNode (*root*)
Update the @clean/@nosent node at root.

readOneAtEditNode (*fn*, *p*)

readOneAtShadowNode (*fn*, *p*, *force=False*)

readOpenFile (*root*, *fileName*, *deleteNodes=False*)
Read an open derived file. Leo 4.5 and later can only read 4.x derived files.

readPostPass (*root*, *thinFile*)
Post-process all vnodes.

readRef (*s*, *i*)
Handle an @<< sentinel.

readStartAll (*s*, *i*)
Read an @+all sentinel.

readStartAt (*s*, *i*)
Read an @+at sentinel.

readStartDoc (*s*, *i*)
Read an @+doc sentinel.

readStartLeo(*s*, *i*)
Read an unexpected @+leo sentinel.

readStartMiddle(*s*, *i*)
Read an @+middle sentinel.

readStartNode(*s*, *i*, *middle=False*)
Read an @+node or @+middle sentinel. This will terminate the previous node.

readStartOthers(*s*, *i*)
Read an @+others sentinel.

readVerbatim(*s*, *i*)
Read an @verbatim sentinel.

read_at_clean_lines(*fn*)
Return all lines of the @clean/@nosent file at fn.

reloadSettings()

rememberReadPath(*fn*, *p*)
Remember the files that have been read *and* the full headline (@<file> type) that caused the read.

remove(*fileName*, *verbose=True*)

removeCommentDelims(*p*)
If the present @language/@comment settings do not specify a single-line comment we remove all block comment delims from h. This prevents headline text from interfering with the parsing of node sentinels.

rename(*src*, *dst*, *mode=None*, *verbose=True*)
Remove dst if it exists, then rename src to dst. Change the mode of the renamed file if mode is given. Return True if all went well.

replaceFileWithString(*fn*, *s*)
Replace the file with s if s is different from theFile's contents.

Return True if theFile was changed.

This is used only by the @shadow logic.

replaceTargetFileIfDifferent(*root*, *ignoreBlankLines=False*)
Create target file as follows: 1. If target file does not exist, rename output file to target file. 2. If target file is identical to output file, remove the output file. 3. If target file is different from output file,

remove target file, then rename output file to be target file.

Return True if the original file was changed.

reportCorrection(*old*, *new*, *v*)
Debugging only. Report changed perfect import lines.

runPyflakes(*root*, *pyflakes_errors_only*)
Run pyflakes on the selected node.

saveOutlineIfPossible()
Save the outline if only persistence data nodes are dirty.

scanAllDirectives(*p*, *forcePythonSentinels=False*, *importing=False*, *issuePathWarning=False*, *reading=False*)
Scan p and p's ancestors looking for directives, setting corresponding AtFile ivars.

scanFirstLines(*firstLines*)
Append all lines before the @+leo line to firstLines.

Empty lines are ignored because empty @first directives are ignored.

We can not call sentinelKind here because that depends on the comment delimiters we set here.

scanForClonedSibs (*parent_v*, *v*)
Scan the siblings of VNode v looking for clones of v. Return the number of cloned sibs and n where p is the n'th cloned sibling.

scanHeader (*fileName*, *giveErrors=True*)
Scan the @+leo sentinel.

Sets self.encoding, and self.start/endSentinelComment.

Returns (firstLines,new_df,isThinDerivedFile) where: firstLines contains all @first lines, new_df is True if we are reading a new-format derived file. isThinDerivedFile is True if the file is an @thin file.

scanHeaderForThin (*fileName*)
Return true if the derived file is a thin file.

This is a kludgy method used only by the import code.

scanText4 (*fileName*, *p*, *verbose=False*)
Scan a 4.x derived file non-recursively.

sentinelDict = {'@+all': 72, '@+at': 41, '@+body': 42, '@+doc': 43, '@+leo': 44,

sentinelKind4 (*s*)
Return the kind of sentinel at s.

sentinelKind4_helper (*s*)

sentinelName (*kind*)
Returns the name of the sentinel for warnings..

setDirtyOrphanBits (*root*)
Setting the orphan and dirty flags tells Leo to write the tree. However, the dirty bits get cleared if we are called from the save command.

setPathUa (*p*, *path*)

setTargetFileName (*root*, *toString*)
Set the target file name for at.write.

shouldDeleteChildren (*root*, *thinFile*)
Return True if we should delete all children before a read.

shouldPromptForDangerousWrite (*fn*, *p*)
Return True if a prompt should be issued when writing p (an @<file> node) to fn.

silentWrite (*root*, *toString=False*)

skipIndent (*s*, *i*, *width*)

skipSentinelStart4 (*s*, *i*)
Skip the start of a sentinel.

skipToEndSentinel (*s*, *i*)
Skip to the end of the sentinel line.

startAfterRef = 80

startAll = 72

startAt = 41

startBody = 42

startClone = 81

startComment = 60

startDelims = 61

startDirective = 62

startDoc = 43

startLeo = 44

startMiddle = 73

startNl = 82

startNode = 45

startNonl = 83

startOthers = 46

startRef = 63

startVerbatim = 64

startVerbatimAfterRef = 65

stat(*fileName*)
Return the access mode of named file, removing any setuid, setgid, and sticky bits.

syntaxError(*p*, *body*)
Report a syntax error.

tabNannyNode(*p*, *body*, *suppress=False*)

terminateBody(*v*, *postPass=False*)
Terminate scanning of body text for node v. Set v.b.

terminateNode(*middle=False*, *postPass=True*, *v=None*)
Set the body text of at.v, and issue warning if it has changed.

This is called as follows:

old sentinels: when handling a @-node sentinel. new sentinels: from the post-pass when v.tempBodyList exists.

validInAtOthers(*p*)
Return True if p should be included in the expansion of the @others directive in the body text of p's parent.

warnAboutOrphandAndIgnoredNodes()

warnOnReadOnlyFile(*fn*)

write(*root*, *kind='@unknown'*, *nosentinels=False*, *toString=False*)
Write a 4.x derived file. root is the position of an @<file> node.

writeAll(*writeAtFileNodesFlag=False*, *writeDirtyAtFileNodesFlag=False*, *toString=False*)
Write @file nodes in all or part of the outline

writeAllHelper(*p*, *root*, *force*, *toString*, *writeAtFileNodesFlag*, *writtenFiles*)
Write one file for the at.writeAll. Do *not* write @auto files unless p == root. This prevents the write-all command from needlessly updating the @persistence data, thereby annoyingly changing the .leo file.

writeAsisNode(*p*)
Write the p's node to an @asis file.

writeAtAutoNodes(*event=None*)
Write all @auto nodes in the selected outline.

writeAtAutoNodesHelper(*toString=False*, *writeDirtyOnly=True*)
Write @auto nodes in the selected outline

writeAtShadowNodes(*event=None*)
Write all @shadow nodes in the selected outline.

writeAtShadowNodesHelper(*toString=False*, *writeDirtyOnly=True*)
Write @shadow nodes in the selected outline

writeDirtyAtAutoNodes(*event=None*)
Write all dirty @auto nodes in the selected outline.

writeDirtyAtShadowNodes(*event=None*)
Write all dirty @shadow nodes in the selected outline.

writeError(*message=None*)
Issue an error while writing an @<file> node.

writeException(*root=None*)

writeFromString(*root*, *s*, *forcePythonSentinels=True*, *useSentinels=True*)
Write a 4.x derived file from a string.

This is at.write specialized for scripting.

writeMissing(*p*, *toString=False*)

writeMissingNode(*p*)

writeOneAtAutoNode(*p*, *force=False*, *toString=False*, *trialWrite=False*)
Write p, an @auto node. File indices *must* have already been assigned.

writeOneAtEditNode(*p*, *toString*, *force=False*)
Write one @edit node.

writeOneAtShadowNode(*p*, *toString*, *force*)
Write p, an @shadow node. File indices *must* have already been assigned.

writeOpenFile(*root*, *nosentinels=False*, *toString=False*, *fromString=''*)
Do all writes except asis writes.

write_at_clean_sentinels(*root*)
Return all lines of the @clean tree as if it were written as an @file node.

writer_for_at_auto(*root*)
A factory returning a writer function for the given kind of @auto directive.

writer_for_ext(*ext*)
A factory returning a writer function for the given file extension.

leo.core.leoAtFile.**atFile**
alias of *leo.core.leoAtFile.AtFile*

1.2.1.6 leoBridge Module

A module to allow full access to Leo commanders from outside Leo.

class leo.core.leoBridge.**BridgeController**(*guiName*, *loadPlugins*, *readSettings*, *silent*, *tracePlugins*, *verbose*)
Bases: object

Creates a way for host programs to access Leo.

adjustSysPath()
Adjust sys.path to enable imports as usual with Leo.

completeFileName(*fileName*)

createFrame(*fileName*)
Create a commander and frame for the given file. Create a new frame if the fileName is empty or non-exisent.

createGui()

globals()
Return a fully initialized leoGlobals module.

initLeo()
Init the Leo app to which this class gives access. This code is based on leo.run().

isOpen()
Return True if the bridge is open.

isValidPython()

openLeoFile(*fileName*)
Open a .leo file, or create a new Leo frame if no fileName is given.

reportDirectories()

leo.core.leoBridge.**controller**(*gui='nullGui'*, *loadPlugins=True*, *readSettings=True*, *silent=False*, *tracePlugins=False*, *verbose=False*)
Create an singleton instance of a bridge controller.

1.2.1.7 leoBridgeTest Module

A module that runs unit tests with the leoBridge module.

All options come from sys.argv. See scan_options for the available options.

Important: Leo's core does not use this module in any way.

leo.core.leoBridgeTest.**main**()
The main line of leoBridgeTest.py.

leo.core.leoBridgeTest.**runUnitTests**(*c*, *g*)
Run all the unit tests from the leoBridge.

leo.core.leoBridgeTest.**scanOptions**()
Handle all options and remove them from sys.argv.

1.2.1.8 leoCache Module

A module encapsulating Leo's file caching

class leo.core.leoCache.**Cacher**(*c=None*)
Bases: object

A class that encapsulates all aspects of Leo's file caching.

checkForChangedNodes(*child_tuple*, *fileName*, *parent_v*)
Update the outline described by child_tuple, including all descendants.

clearAllCaches()
Clear the Cachers *only* for all open windows. This is much safer than killing all db's.

clearCache()
Clear the cache for the open window.

collectChangedNodes(*root_v*, *aList*, *fileName*)
Populates c.nodeConflictList with data about nodes that are going to change during recreation of outline from cached list.

commit(*close=True*)

createOutlineFromCacheList(*parent_v*, *aList*, *fileName*, *top=True*)
Create outline structure from recursive aList built by makeCacheList.

createOutlineFromCacheList2(*parent_v*, *aList*)
Create outline structure from recursive aList built by makeCacheList.

fastAddLastChild(*fileName*, *gnxString*, *parent_v*)
Create new VNode as last child of the receiver. If the gnx exists already, create a clone instead of new VNode.

fileKey(*fileName*, *content*, *requireEncodedString=False*)
Compute the hash of fileName and content. fileName may be unicode, content must be bytes (or plain string in Python 2.x).

getCachedGlobalFileRatios()

getCachedStringPosition()

getCachedWindowPositionDict(*fn*)
Return a dict containing window positions.

initFileDB(*fn*)

initGlobalDB()

makeCacheList(*p*)
Create a recursive list describing a tree for use by createOutlineFromCacheList.

readFile(*fileName*, *root*)
Read the file from the cache if possible. Return (s,ok,key)

reportIfNodeChanged(*child_tuple*, *child_v*, *fileName*, *parent_v*)
Schedule a recovered node if child_v is substantially different from an earlier version.

Issue a (rare) warning if two different files are involved.

save(*fn*, *changeName*)

setCachedGlobalsElement(*fn*)

setCachedStringPosition(*str_pos*)

test()

warning(*s*)
Print a warning message in red.

writeFile(*p*, *fileKey*)
Update the cache after reading the file.

class leo.core.leoCache.**PickleShareDB**(*root*)
Bases: object

The main 'connection' object for PickleShare database

clear(*verbose=False*)

get(*key*, *default=None*)

has_key(*key*)

items()

keys(*globpat=None*)
Return all keys in DB, or all keys matching a glob

uncache(**items*)
Removes all, or specified items from cache

Use this after reading a large amount of large objects to free up memory, when you won't be needing the objects for a while.

class leo.core.leoCache.**SqlitePickleShare**(*root*)
Bases: object

The main 'connection' object for SqlitePickleShare database

clear(*verbose=False*)

get(*key*, *default=None*)

has_key(*key*)

init_dbtables(*conn*)

items()

keys(*globpat=None*)
Return all keys in DB, or all keys matching a glob

reset_protocol_in_values()

uncache(**items*)
not used in SqlitePickleShare

1.2.1.9 leoChapters Module

Classes that manage chapters in Leo's core.

class leo.core.leoChapters.**Chapter**(*c*, *chapterController*, *name*)
Bases: object

A class representing the non-gui data of a single chapter.

chapterSelectHelper(*w=None*, *selectEditor=True*)

findEditorInChapter(*p*)
return w, an editor displaying position p.

findPositionInChapter(*p1*, *strict=False*)
Return a valid position p such that p.v == v.

findRootNode()
Return the @chapter node for this chapter.

positionIsInChapter(*p*)

select(*w=None*, *selectEditor=True*)
Restore chapter information and redraw the tree when a chapter is selected.

unselect()
Remember chapter info when a chapter is about to be unselected.

class leo.core.leoChapters.**ChapterController**(*c*)
Bases: object

A per-commander controller that manages chapters and related nodes.

backChapter(*event=None*)

cmd()
Command decorator for the ChapterController class.

createIcon()
Create chapter-selection Qt ListBox in the icon area.

error(*s*)

findAnyChapterNode()
Return True if the outline contains any @chapter node.

findChapterNameForPosition(*p*)
Return the name of a chapter containing p or None if p does not exist.

findChapterNode(*name*)
Return the position of the first @chapter node with the given name anywhere in the entire outline.

All @chapter nodes are created as children of the @chapters node, but users may move them anywhere.

finishCreate()
Create the box in the icon area.

getChapter(*name*)

getSelectedChapter()

inChapter()

makeCommand(*chapterName*, *binding=None*)
Make chapter-select-<chapterName> command.

nextChapter(*event=None*)

note(*s*, *killUnitTest=False*)

parseHeadline(*p*)
Return the chapter name and key binding for p.h.

reloadSettings()

sanitize(*s*)
Convert s to a safe chapter name.

selectChapter(*event=None*)
Use the minibuffer to get a chapter name, then create the chapter.

selectChapter1(*event*)

selectChapterByName(*name*, *collapse=True*)
Select a chapter. Return True if a redraw is needed.

selectChapterByNameHelper(*chapter*, *collapse=True*)
Select the chapter, and redraw if necessary.

selectChapterForPosition(*p*, *chapter=None*)
Select a chapter containing position p. New in Leo 4.11: prefer the given chapter if possible. Do nothing if p if p does not exist or is in the presently selected chapter.

Note: this code calls c.redraw() if the chapter changes.

setAllChapterNames()
Called early and often to discover all chapter names.

warning(*s*)

1.2.1.10 leoColor Module

A color database for Leo.

leo_color_database is a dictionary of color names mapped onto the colors '#rrggbb' representation.

The color names are derived from standard Linux color names which includes all Tk color names.

The names have been normalized by excluding spaces and removing capitalization. This should also be done for all new colors.

Accessor functions are provided which will normalize name requests before looking them up in the database.

These are:

```
getColor (aka: get)
getColorRGB (aka: getRGB)
getColorCairo (aka: getCairo)
```

Use these functions as follows:

```
import leo.core.leoColor as leoColor
leoColor.getRGB(name, default)
```

If neither 'name' nor 'default' can be translated then accessor functions will return None.

leo.core.leoColor.**get**(*name*, *default=None*)
Translate a named color into #rrggbb' format.

if 'name' is not a string it is returned unchanged.

If 'name' is already in '#rrggbb' format then it is returned unchanged.

If 'name' is not in global_color_database then getColor(default, None) is called and that result returned.

leo.core.leoColor.**getCairo**(*name*, *default=None*)
Convert a named color into a cairo color tuple.

leo.core.leoColor.**getColor**(*name*, *default=None*)
Translate a named color into #rrggbb' format.

if 'name' is not a string it is returned unchanged.

If 'name' is already in '#rrggbb' format then it is returned unchanged.

If 'name' is not in global_color_database then getColor(default, None) is called and that result returned.

leo.core.leoColor.**getColorCairo**(*name*, *default=None*)
Convert a named color into a cairo color tuple.

leo.core.leoColor.**getColorRGB**(*name*, *default=None*)
Convert a named color into an (r, g, b) tuple.

leo.core.leoColor.**getRGB**(*name*, *default=None*)
Convert a named color into an (r, g, b) tuple.

1.2.1.11 leoCommands Module

class leo.core.leoCommands.**Commands**(*fileName*, *relativeFileName=None*, *gui=None*, *previousSettings=None*)

Bases: object

A per-outline class that implements most of Leo's commands. The "c" predefined object is an instance of this class.

c.initObjects() creates sucommanders corresponding to files in the leo/core and leo/commands. All of Leo's core code is accessible via this class and its subcommanders.

g.app.pluginsController is Leo's plugins controller. Many plugins inject controllers objects into the Commands class. These are another kind of subcommander.

The @g..commander_command decorator injects methods into this class.

BeginUpdate()
Deprecated: does nothing.

BringToFront(*c2=None*, *set_focus=True*)

EndUpdate(*flag=True*)
Request a redraw of the screen if flag is True.

add_command(*menu*, ***keys*)

alert(*message*)

allNodes_iter()
A generator return all positions of the outline, in outline order.

all_nodes()
A generator returning all vnodes in the outline, in outline order.

all_positions()
A generator return all positions of the outline, in outline order.

all_positions_iter()
A generator return all positions of the outline, in outline order.

all_positions_with_unique_tnodes_iter()
A generator return all positions of the outline, in outline order. Returns only the first position for each vnode.

all_positions_with_unique_vnodes_iter()
A generator return all positions of the outline, in outline order. Returns only the first position for each vnode.

all_roots(*predicate=None*)
A generator yielding *all* the root positions in the outline that satisfy the given predicate. p.isAnyAtFileNode is the default predicate.

The generator yields all **root** anywhere in the outline that satisfy the predicate. Once a root is found, the generator skips its subtree.

all_tnodes_iter()
A generator returning all vnodes in the outline, in outline order.

all_unique_nodes()
A generator returning each vnode of the outline.

all_unique_positions()
A generator return all positions of the outline, in outline order. Returns only the first position for each vnode.

all_unique_roots(*predicate=None*)
A generator yielding all unique root positions in the outline that satisfy the given predicate. p.isAnyAtFileNode is the default predicate.

The generator yields all **root** anywhere in the outline that satisfy the predicate. Once a root is found, the generator skips its subtree.

all_unique_tnodes_iter()
A generator returning each vnode of the outline.

all_unique_vnodes_iter()
A generator returning each vnode of the outline.

all_vnodes_iter()
A generator returning all vnodes in the outline, in outline order.

appendStringToBody(*p*, *s*)

backup(*fileName=None*, *prefix=None*, *silent=False*, *useTimeStamp=True*)
Back up given fileName or c.fileName(). If useTimeStamp is True, append a timestamp to the filename.

backup_helper(*base_dir=None*, *env_key='LEO_BACKUP'*, *sub_dir=None*, *use_git_prefix=True*)
A helper for scripts that back up a .leo file. Use os.environ[env_key] as the base_dir only if base_dir is not given. Backup to base_dir or join(base_dir, sub_dir).

beginUpdate()
Deprecated: does nothing.

bodyWantsFocus()

bodyWantsFocusNow()

bringToFront(*c2=None*, *set_focus=True*)

canClone()

canContractAllHeadlines()
Contract all nodes in the tree.

canContractAllSubheads()

canContractParent()

canContractSubheads()

canCutOutline()

canDehoist()
Return True if do-hoist should be enabled in a menu. Should not be used in any other context.

canDeleteHeadline()

canDemote()

canExpandAllHeadlines()
Return True if the Expand All Nodes menu item should be enabled.

canExpandAllSubheads()

canExpandSubheads()

canExtract()

canExtractSection()

canExtractSectionNames()

canFindMatchingBracket()

canGoToNextDirtyHeadline()

canGoToNextMarkedHeadline()

canHoist()
Return True if hoist should be enabled in a menu. Should not be used in any other context.

canMarkChangedHeadlines()

canMarkChangedRoots()

canMoveOutlineDown()

canMoveOutlineLeft()

canMoveOutlineRight()

canMoveOutlineUp()

canPasteOutline(*s=None*)

canPromote()

canRedo()

canSelectThreadBack()

canSelectThreadNext()

canSelectVisBack()

canSelectVisNext()

canShiftBodyLeft()

canShiftBodyRight()

canSortChildren()

canSortSiblings()

canUndo()

canUnmarkAll()

checkAllPythonCode(*event=None*, *unittest=False*, *ignoreAtIgnore=True*)
Check all nodes in the selected tree for syntax and tab errors.

checkBatchOperationsList(*aList*)

checkDrag(*root*, *target*)
Return False if target is any descendant of root.

checkFileTimeStamp(*fn*)
Return True if the file given by fn has not been changed since Leo read it or if the user agrees to overwrite it.

checkGnxs()
Check the consistency of all gnx's and remove any tnodeLists. Reallocate gnx's for duplicates or empty gnx's. Return the number of structure_errors found.

checkLinks()
Check the consistency of all links in the outline.

checkMoveWithParentWithWarning(*root*, *parent*, *warningFlag*)
Return False if root or any of root's descendents is a clone of parent or any of parents ancestors.

checkOutline(*event=None*, *check_links=False*)
Check for errors in the outline. Return the count of serious structure errors.

checkParentAndChildren(*p*)
Check consistency of parent and child data structures.

checkPythonCode(*event=None*, *unittest=False*, *ignoreAtIgnore=True*, *suppressErrors=False*, *checkOnSave=False*)
Check the selected tree for syntax and tab errors.

checkPythonNode(*p*, *unittest=False*, *suppressErrors=False*)

checkSiblings(*p*)
Check the consistency of next and back links.

checkThreadLinks(*p*)
Check consistency of threadNext & threadBack links.

check_event(*event*)
Check an event object.

clearAllMarked()

clearAllVisited()

clearMarked(*p*)

cloneFindByPredicate(*generator*, *predicate*, *failMsg=None*, *flatten=False*, *iconPath=None*, *redraw=True*, *undoType=None*)
Traverse the tree given using the generator, cloning all positions for which predicate(p) is True. Undoably move all clones to a new node, created as the last top-level node. Returns the newly-created node. Arguments:

generator, The generator used to traverse the tree. predicate, A function of one argument p returning true if p should be included. failMsg=None, Message given if nothing found. Default is no message. flatten=False, True: Move all node to be parents of the root node. iconPath=None, Full path to icon to attach to all matches. redraw=True, True: redraw the screen. undo_type=None, The undo/redo name shown in the Edit:Undo menu.

The default is 'clone-find-predicate'

command_count = 0

computeWindowTitle(*fileName*)
Set the window title and fileName.

contractAllHeadlines(*event=None*, *redrawFlag=True*)
Contract all nodes in the outline.

contractSubtree(*p*)

createCloneFindPredicateRoot(*flatten*, *undoType*)
Create a root node for clone-find-predicate.

createCommandNames()
Create all entries in c.commandsDict. Do *not* clear c.commandsDict here.

createNodeFromExternalFile(*fn*)
Read the file into a node. Return None, indicating that c.open should set focus.

createNodeHierarchy(*heads*, *parent=None*, *forcecreate=False*)
Create the proper hierarchy of nodes with headlines defined in 'heads' under 'parent'

params: parent - parent node to start from. Set to None for top-level nodes heads - list of headlines in order to create, i.e. ['foo','bar','baz']

will create: parent -foo –bar —baz

forcecreate - If False (default), will not create nodes unless they don't exist If True, will create nodes regardless of existing nodes

returns the final position ('baz' in the above example)

currentPosition()
Return a copy of the presently selected position or a new null position. So c.p.copy() is never necessary.

currentPositionHasNext()
Return True if the current position is the root position.

This method is called during idle time, so not generating positions here fixes a major leak.

currentPositionIsRootPosition()
Return True if the current position is the root position.

This method is called during idle time, so not generating positions here fixes a major leak.

currentVnode()
Return a copy of the presently selected position or a new null position. So c.p.copy() is never necessary.

deletePositionsInList(*aList*, *callback=None*, *redraw=True*)
Delete all vnodes corresponding to the positions in aList. If a callback is given, the callback is called for every node in the list.

The callback takes one explicit argument, p. As usual, the callback can bind values using keyword arguments.

This is *very* tricky code. The theory of operation section explains why.

diff_file(*fn*, *rev1='HEAD'*, *rev2=''*, *directory=None*)
Create an outline describing the git diffs for all files changed between rev1 and rev2.

diff_two_branches(*branch1*, *branch2*, *fn*, *directory=None*)
Create an outline describing the git diffs for all files changed between rev1 and rev2.

diff_two_revs(*directory=None*, *rev1=''*, *rev2=''*)
Create an outline describing the git diffs for all files changed between rev1 and rev2.

disable_redraw()
Disable all redrawing until enabled.

doBatchOperations(*aList=None*)

doCommand(*command*, *label*, *event=None*)
Execute the given command, invoking hooks and catching exceptions.

The code assumes that the "command1" hook has completely handled the command if g.doHook("command1") returns False. This provides a simple mechanism for overriding commands.

dragAfter(*p*, *after*)

dragCloneAfter(*p*, *after*)

dragCloneToNthChildOf(*p*, *parent*, *n*)

dragToNthChildOf(*p*, *parent*, *n*)

dumpPosition(*p*)
Dump position p and it's ancestors.

edit_widget(*p*)

enable_redraw()

endEditing()

endUpdate(*flag=True*)
Request a redraw of the screen if flag is True.

executeAnyCommand(*command*, *event*)
Execute a command, no matter how defined.

Supports @g.commander_command and @g.new_cmd_decorator and plain methods.

executeMinibufferCommand(*commandName*)

executeScript(*event=None*, *args=None*, *p=None*, *script=None*, *useSelectedText=True*, *define_g=True*, *define_name='__main__'*, *silent=False*, *namespace=None*, *raiseFlag=False*)
Execute a *Leo* script. Keyword args: args=None Not None: set script_args in the execution environment. p=None Get the script from p.b, unless script is given. script=None None: use script in p.b or c.p.b useSelectedText=True False: use all the text in p.b or c.p.b. define_g=True True: define g for the script. define_name='__main__' Not None: define the name symbol. silent=False No longer used. namespace=None Not None: execute the script in this namespace. raiseFlag=False True: reraise any exceptions.

executeScriptHelper(*args*, *define_g*, *define_name*, *namespace*, *script*)

expandAllAncestors(*p*)
Expand all ancestors without redrawing. Return a flag telling whether a redraw is needed.

expandSubtree(*v*)

expandToLevel(*level*)

fileName()

findNodeOutsideAnyAtFileTree(*target*)
Select the first clone of target that is outside any @file node.

findRootPosition()
Return the root position.

Root position is the first position in the document. Other top level positions are siblings of this node.

find_b(*regex*, *flags=10*)
Return list (a PosList) of all nodes whose body matches regex one or more times.

find_h(*regex*, *flags=2*)
Return list (a PosList) of all nodes where zero or more characters at the beginning of the headline match regex

finishCreate()
Finish creating the commander and all sub-objects. This is the last step in the startup process.

firstVisible()
Move to the first visible node of the present chapter or hoist.

force_redraw(*p=None*, *setFocus=False*)
Redraw the screen immediately.

getBodyLines(*expandSelection=False*)
Return head,lines,tail where:

before is string containg all the lines before the selected text (or the text before the insert point if no selection) lines is a list of lines containing the selected text (or the line containing the insert point if no selection) after is a string all lines after the selected text (or the text after the insert point if no selection)

getLanguageAtCursor(*p*, *language*)
Return the language in effect at the present insert point. Use the language argument as a default if no @language directive seen.

getNodeFileName(*p*)
Return the full file name at node p, including effects of all @path directives. Return None if p is no kind of @file node.

getNodePath(*p*)
Return the path in effect at node p.

getSelectedPositions()
Get list (PosList) of currently selected positions

So far only makes sense on qt gui (which supports multiselection)

getTabWidth(*p*)
Return the tab width in effect at p.

getTime(*body=True*)

get_focus()

get_requested_focus()

git_diff(*rev1='HEAD'*, *rev2=''*, *directory=None*)

goToLineNumber(*n*)
Go to line n (zero-based) of a script. A convenience method called from g.handleScriptException.

goToScriptLineNumber(*n*, *p*)
Go to line n (zero-based) of a script. A convenience method called from g.handleScriptException.

hasAmbiguousLanguage(*p*)
Return True if p.b contains different @language directives.

hash()

idle_focus_count = 0

idle_focus_helper(*tag*, *keys*)
An idle-tme handler that ensures that focus is *somewhere*.

initAfterLoad()
Provide an offical hook for late inits of the commander.

initCommandIvars()
Init ivars used while executing a command.

initConfigSettings()
Init all cached commander config settings.

initDebugIvars()
Init Commander debugging ivars.

initDocumentIvars()
Init per-document ivars.

initEventIvars()
Init ivars relating to gui events.

initFileIvars(*fileName*, *relativeFileName*)
Init file-related ivars of the commander.

initObjectIvars()

initObjects(*gui*)

initOptionsIvars()
Init Commander ivars corresponding to user options.

initSettings(*previousSettings*)
Init the settings *before* initing the objects.

init_error_dialogs()

initialFocusHelper()

interactive(*callback*, *event*, *prompts*)
c.interactive: Prompt for up to three arguments from the minibuffer.

The number of prompts determines the number of arguments.

Use the @command decorator to define commands. Examples:

@g.command('i3') def i3_command(event):

c = event.get('c') if not c: return

def callback(args, c, event): g.trace(args) c.bodyWantsFocus()

c.interactive(callback, event, prompts=['Arg1: ', ' Arg2: ', ' Arg3: '])

interactive1(*callback*, *event*, *prompts*)

interactive2(*callback*, *event*, *prompts*)

interactive3(*callback*, *event*, *prompts*)

invalidateFocus()
Indicate that the focus is in an invalid location, or is unknown.

isChanged()

isCurrentPosition(*p*)

isRootPosition(*p*)

is_unusual_focus(*w*)
Return True if w is not in an expected place.

lastTopLevel()
Return the last top-level position in the outline.

lastVisible()
Move to the last visible node of the present chapter or hoist.

last_unusual_focus = None

logWantsFocus()

logWantsFocusNow()

looksLikeDerivedFile(*fn*)
Return True if fn names a file that looks like an external file written by Leo.

markAllAtFileNodesDirty(*event=None*)
Mark all @file nodes as changed.

markAtFileNodesDirty(*event=None*)
Mark all @file nodes in the selected tree as changed.

minibufferWantsFocus()

minibufferWantsFocusNow()

navHelper(*p*, *ch*, *extend*)

navQuickKey()
return true if there are two quick outline navigation keys in quick succession.

Returns False if @float outline_nav_extend_delay setting is 0.0 or unspecified.

notValidInBatchMode(*commandName*)

nullPosition()
New in Leo 5.5: Return None. Using empty positions masks problems in program logic.

In fact, there are no longer any calls to this method in Leo's core.

onCanvasKey(*event*)
Navigate to the next headline starting with ch = event.char. If ch is uppercase, search all headlines; otherwise search only visible headlines. This is modelled on Windows explorer.

openWith(*event=None*, *d=None*)
This is *not* a command.

Handles the items in the Open With... menu.

See ExternalFilesController.open_with for details about d.

os_path_finalize(*path*, ***keys*)

os_path_finalize_join(**args*, ***keys*)

outerUpdate()
Handle delayed focus requests and modified events.

p
commander current position property

positionExists(*p*, *root=None*, *trace=False*)
Return True if a position exists in c's tree

printCommandsDict()

putHelpFor(*s*, *short_title=''*)
Helper for various help commands.

raise_error_dialogs(*kind='read'*)
Warn abouit read/write failures.

recolor(***kwargs*)

recolorCommand(*event=None*)
Force a full recolor.

recolor_now(***kwargs*)

recreateGnxDict()
Recreate the gnx dict prior to refreshing nodes from disk.

recursiveImport(*dir_*, *kind*, *add_path=True*, *recursive=True*, *safe_at_file=True*, *theTypes=None*)
Recursively import all python files in a directory and clean the results.

Parameters:: dir_ The root directory or file to import. kind One of ('@clean','@edit','@file','@nosent'). add_path=True True: add a full @path directive to @<file> nodes. recursive=True True: recurse into subdirectories. safe_at_file=True True: produce @@file nodes instead of @file nodes. theTypes=None A list of file extensions to import.

None is equivalent to ['.py']

This method cleans imported files as follows:

- Replace backslashes with forward slashes in headlines.
- Remove empty nodes.
- Add @path directives that reduce the needed path specifiers in descendant nodes.
- Add @file to nodes or replace @file with @@file.

redirectScriptOutput()

redraw(*p=None*, *setFocus=False*)
Redraw the screen immediately.

redrawAndEdit(*p*, *selectAll=False*, *selection=None*, *keepMinibuffer=False*)
Redraw the screen and edit p's headline.

redraw_after_contract(*p=None*, *setFocus=False*)

redraw_after_expand(*p=None*, *setFocus=False*)

redraw_after_head_changed()
Redraw the screen (if needed) when editing ends. This may be a do-nothing for some gui's.

redraw_after_icons_changed()
Update the icon for the presently selected node

redraw_after_select(*p*)
Redraw the screen after node p has been selected.

redraw_later()
Ensure that c.redraw() will be called eventually.

c.outerUpdate will call c.redraw() only if no other code calls c.redraw().

redraw_now(*p=None*, *setFocus=False*)
Redraw the screen immediately.

registerReloadSettings(*obj*)
Enter object into c.configurables.

relativeFileName()

reloadConfigurableSettings()
Call all reloadSettings method in c.subcommanders, c.configurables and other known classes.

request_focus(*w*)

rootPosition()
Return the root position.

Root position is the first position in the document. Other top level positions are siblings of this node.

rootVnode()
Return the root position.

Root position is the first position in the document. Other top level positions are siblings of this node.

safe_all_positions()
A generator returning all positions of the outline. This generator does *not* assume that vnodes are never their own ancestors.

scanAllDirectives(*p=None*)
Scan p and ancestors for directives.

Returns a dict containing the results, including defaults.

scanAtPathDirectives(*aList*)
Scan aList for @path directives. Return a reasonable default if no @path directive is found.

scanAtRootDirectives(*aList*)
Scan aList for @root-code and @root-doc directives.

selectPosition(*p*, ***kwargs*)
Select a new position, redrawing the screen *only* if we must change chapters.

selectVnode(*p*, ***kwargs*)
Select a new position, redrawing the screen *only* if we must change chapters.

setBodyString(*p*, *s*)

setChanged(*changedFlag=True*, *redrawFlag=True*)
Set or clear the marker that indicates that the .leo file has been changed.

setCloneFindByPredicateIcon(*iconPath*, *p*)
Attach an icon to p.v.u.

setComplexCommand(*commandName*)
Make commandName the command to be executed by repeat-complex-command.

setCurrentDirectoryFromContext(*p*)

setCurrentPosition(*p*)
Set the presently selected position. For internal use only. Client code should use c.selectPosition instead.

setCurrentVnode(*p*)
Set the presently selected position. For internal use only. Client code should use c.selectPosition instead.

setFileTimeStamp(*fn*)
Update the timestamp for fn..

setHeadString(*p*, *s*)
Set the p's headline and the corresponding tree widget to s.

This is used in by unit tests to restore the outline.

setLog()

setMarked(*p*)

setPositionAfterSort(*sortChildren*)

setRootPosition(*unused_p=None*)
Set c._rootPosition.

setRootVnode(*v*)

setTopPosition(*p*)
Set the root positioin.

setTopVnode(*p*)
Set the root positioin.

setWindowPosition()

set_focus(*w*, *force=False*)

shortFileName()

shortFilename()

shouldBeExpanded(*p*)
Return True if the node at position p should be expanded.

syntaxErrorDialog()
Warn about syntax errors in files.

tabNannyNode(*p*, *headline*, *body*, *unittest=False*, *suppressErrors=False*)
Check indentation using tabnanny.

topPosition()
Return the root position.

topVnode()
Return the root position.

traceFocus(*w*)

trace_idle_focus(*w*)
Trace the focus for w, minimizing chatter.

treeFocusHelper()

treeSelectHelper(*p*)

treeWantsFocus()

treeWantsFocusNow()

trimTrailingLines(*p*)
Trims trailing blank lines from a node.

It is surprising difficult to do this during Untangle.

universalCallback(*source_c*, *function*)
Create a universal command callback.

Create and return a callback that wraps a function with an rClick signature in a callback which adapts standard minibufer command callbacks to a compatible format.

This also serves to allow rClick callback functions to handle minibuffer commands from sources other than rClick menus so allowing a single function to handle calls from all sources.

A function wrapped in this wrapper can handle rclick generator and invocation commands and commands typed in the minibuffer.

It will also be able to handle commands from the minibuffer even if rclick is not installed.

universallCallback(*source_c*, *function*)
Create a universal command callback.

Create and return a callback that wraps a function with an rClick signature in a callback which adapts standard minibufer command callbacks to a compatible format.

This also serves to allow rClick callback functions to handle minibuffer commands from sources other than rClick menus so allowing a single function to handle calls from all sources.

A function wrapped in this wrapper can handle rclick generator and invocation commands and commands typed in the minibuffer.

It will also be able to handle commands from the minibuffer even if rclick is not installed.

unredirectScriptOutput()

updateBodyPane(*head*, *middle*, *tail*, *undoType*, *oldSel*, *oldYview*, *preserveSel=False*)
Handle changed text in the body pane.

updateSyntaxColorer(*v*)

validateOutline(*event=None*)

visLimit()
Return the topmost visible node. This is affected by chapters and hoists.

vnode2allPositions(*v*)
Given a VNode v, find all valid positions p such that p.v = v.

Not really all, just all for each of v's distinct immediate parents.

vnode2position(*v*)
Given a VNode v, construct a valid position p such that p.v = v.

widgetWantsFocus(*w*)

widgetWantsFocusNow(*w*)

widget_name(*widget*)

writeScriptFile(*script*)

leo.core.leoCommands.**cmd**(*name*)
Command decorator for the Commands class.

1.2.1.12 leoCompare Module

Leo's base compare class.

class leo.core.leoCompare.**BaseLeoCompare**(*commands=None*, *appendOutput=False*, *ignoreBlankLines=True*, *ignoreFirstLine1=False*, *ignoreFirstLine2=False*, *ignoreInteriorWhitespace=False*, *ignoreLeadingWhitespace=True*, *ignoreSentinelLines=False*, *limitCount=0*, *limitToExtension='.py'*, *makeWhitespaceVisible=True*, *printBothMatches=False*, *printMatches=False*, *printMismatches=True*, *printTrailingMismatches=False*, *outputFileName=None*)
Bases: object

The base class for Leo's compare code.

compare_directories(*path1*, *path2*)

compare_files(*name1*, *name2*)

compare_lines(*s1*, *s2*)

compare_list_of_files(*aList1*)

compare_open_files(*f1*, *f2*, *name1*, *name2*)

compare_two_files(*name1*, *name2*)
A helper function.

doOpen(*name*)

dump(*tag*, *s*)

dumpToEndOfFile(*tag*, *f*, *s*, *line*, *printTrailing*)

filecmp(*f1*, *f2*)

isLeoHeader(*s*)

isSentinel(*s*, *sentinelComment*)

openOutputFile()

show(*s*)

showIvars()

class leo.core.leoCompare.**CompareLeoOutlines**(*c*)
Bases: object

A class to do outline-oriented diffs of two or more .leo files. Similar to GitDiffController, adapted for use by scripts.

compute_dicts(*c1*, *c2*)
Compute inserted, deleted, changed dictionaries.

create_compare_node(*c1*, *c2*, *d*, *kind*)
Create nodes describing the changes.

create_file_node(*diff_list*, *fn1*, *fn2*)
Create an organizer node for the file.

create_root(*aList*)
Create the top-level organizer node describing all the diffs.

diff_list_of_files(*aList*, *visible=True*)
The main entry point for scripts.

diff_two_files(*fn1*, *fn2*)
Create an outline describing the git diffs for fn.

find_gnx(*c*, *gnx*)
Return a position in c having the given gnx.

finish()
Finish execution of this command.

get_file(*path*)
Return the contents of the file whose path is given.

make_diff_outlines(*c1*, *c2*)
Create an outline-oriented diff from the outlines c1 and c2.

open_outline(*fn*)
Find the commander for fn, creating a new outline tab if necessary.

Using open commanders works because we always read entire .leo files.

class leo.core.leoCompare.**LeoCompare**(*commands=None*, *appendOutput=False*, *ignoreBlankLines=True*, *ignoreFirstLine1=False*, *ignoreFirstLine2=False*, *ignoreInteriorWhitespace=False*, *ignoreLeadingWhitespace=True*, *ignoreSentinelLines=False*, *limitCount=0*, *limitToExtension='.py'*, *makeWhitespaceVisible=True*, *printBothMatches=False*, *printMatches=False*, *printMismatches=True*, *printTrailingMismatches=False*, *outputFileName=None*)

Bases: *leo.core.leoCompare.BaseLeoCompare*

A class containing Leo's compare code.

These are not very useful comparisons.

leo.core.leoCompare.**diffMarkedNodes**(*event*)
When two or more nodes are marked, this command does the following:

- Creates a "diff marked node" as the last top-level node. The body of this node contains "diff n" nodes, one for each pair of compared nodes.
- Each diff n contains the diffs between the two diffed nodes, that is, difflib.Differ().compare(p1.b, p2.b). The children of the diff n are *clones* of the two compared nodes.

leo.core.leoCompare.**diff_and_open_leo_files**(*event*)
Open a dialog prompting for two or more .leo files.

Opens all the files and creates a top-level node in c's outline showing the diffs of those files, two at a time.

leo.core.leoCompare.**diff_leo_files**(*event*)
Open a dialog prompting for two or more .leo files.

Creates a top-level node showing the diffs of those files, two at a time.

leo.core.leoCompare.**diff_leo_files_helper**(*event*, *title*, *visible*)
Prompt for a list of .leo files to open.

leo.core.leoCompare.**go**()

1.2.1.13 leoConfig Module

1.2.1.14 leoDebugger Module

Per-commander debugging class.

class leo.core.leoDebugger.**leoDebugger**(*c*)
Bases: pdb.Pdb

Leo's debugger class: a thin wrapper around Python's pdb class.

leo.core.leoDebugger.**set_trace**(*c*)

1.2.1.15 leoDynamicTest Module

A module to run unit tests with the leoBridge module. Leo's unit test code uses this module when running unit tests externally.

leo.core.leoDynamicTest.**main**()
Run a dynamic test using the Leo bridge.

leo.core.leoDynamicTest.**runUnitTests**(*c*, *g*)

leo.core.leoDynamicTest.**scanOptions**()
Handle all options and remove them from sys.argv.

1.2.1.16 leoEditCommands Module

1.2.1.17 leoFileCommands Module

Classes relating to reading and writing .leo files.

exception leo.core.leoFileCommands.**BadLeoFile**(*message*)
Bases: exceptions.Exception

class leo.core.leoFileCommands.**FileCommands**(*c*)
Bases: object

A class creating the FileCommands subcommander.

archivedPositionToPosition(*s*)

assignFileIndices()
Assign a file index to all tnodes

bytes_to_unicode(*ob*)
recursively convert bytes objects in strings / lists / dicts to str objects, thanks to TNT http://stackoverflow.com/questions/22840092/unpickling-data-from-python-2-with-unicode-strings-in-python-3

Needed for reading Python 2.7 pickles in Python 3.4 in getSaxUa()

canonicalTnodeIndex(*index*)
Convert Tnnn to nnn, leaving gnx's unchanged.

checkLeoFile(*event=None*)
The check-leo-file command.

checkPaste(*parent*, *p*)
Return True if p may be pasted as a child of parent.

cleanSaxInputString(*s*)
Clean control characters from s. s may be a bytes or a (unicode) string.

cmd()
Command decorator for the FileCommands class.

compactFileIndices()
Assign a file index to all tnodes

compute_attribute_bits(*forceWrite*, *p*)
Return the initial values of v's attributes.

createActualFile(*fileName*, *toOPML*, *toZip*)

createBackupFile(*fileName*)
Create a closed backup file and copy the file to it, but only if the original file exists.

createSaxChildren(*sax_node*, *parent_v*)
Create vnodes for all children in sax_node.children.

createSaxVnode(*sax_node*, *parent_v*)
Create a vnode, or use an existing vnode.

createUaList(*aList*)
Given aList of pairs (p,torv), return a list of pairs (torv,d) where d contains all picklable items of torv.unknownAttributes.

decodePosition(*s*)
Creates position from its string representation encoded by fc.encodePosition.

deleteFileWithMessage(*fileName*, *unused_kind*)

dumpSaxTree(*root*, *dummy*)

encodePosition(*p*)
New schema for encoding current position hopefully simplier one.

exportDbVersion(*conn*)

exportGeomToSqlite(*conn*)

exportHashesToSqlite(*conn*)

exportToSqlite(*fileName*)
Dump all vnodes to sqlite database. Returns True on success.

exportVnodesToSqlite(*conn*, *rows*)

getDescendentAttributes(*s*, *tag=''*)
s is a list of gnx's, separated by commas from a <v> or <t> element. Parses s into a list.

This is used to record marked and expanded nodes.

getDescendentUnknownAttributes(*s*, *v=None*)
Unhexlify and unpickle t/v.descendentUnknownAttribute field.

getLeoFile(*theFile*, *fileName*, *readAtFileNodesFlag=True*, *silent=False*, *checkOpenFiles=True*)
Read a .leo file. The caller should follow this with a call to c.redraw().

getLeoFileHelper(*theFile*, *fileName*, *silent*)
Read the .leo file and create the outline.

getLeoOutline(*s*, *reassignIndices=True*)
Read a Leo outline from string s in clipboard format.

getLeoOutlineFromClipboard(*s*, *reassignIndices=True*)
Read a Leo outline from string s in clipboard format.

getPosFromClipboard(*s*)

getSaxUa(*attr*, *val*, *kind=None*)
Parse an unknown attribute in a <v> or <t> element. The unknown tag has been pickled and hexlify'd.

getVnodeFromClipboard(*s*)

getWindowGeometryFromDb(*conn*)

handleNodeConflicts()
Create a 'Recovered Nodes' node for each entry in c.nodeConflictList.

handleTnodeSaxAttributes(*sax_node*, *v*)

handleVnodeSaxAttributes(*sax_node*, *v*)
The native attributes of <v> elements are a, t, vtag, tnodeList, marks, expanded, and descendentTnode/VnodeUnknownAttributes.

handleWriteLeoFileException(*fileName*, *backupName*, *theActualFile*)

initIvars()
Init ivars of the FileCommands class.

initNewDb(*conn*)
Initializes tables and returns None

initReadIvars()

isReadOnly(*fileName*)

linkChildrenToParents(*p*)
Populate the parent links in all children of p.

openLeoFile(*theFile*, *fileName*, *readAtFileNodesFlag=True*, *silent=False*)
Open a Leo file.

parse_leo_file(*theFile*, *inputFileName*, *silent*, *inClipboard*, *s=None*)

pickle(*torv*, *val*, *tag*)
Pickle val and return the hexlified result.

prepareDbTables(*conn*)

propegateDirtyNodes()

put(*s*)
Put string s to self.outputFile. All output eventually comes here.

putClipboardHeader()

putDescendentAttributes(*p*)

putDescendentVnodeUas(*p*)
Return the a uA field for descendent VNode attributes, suitable for reconstituting uA's for anonymous vnodes.

putFindSettings()

putGlobals()

putHeader()

putLeoFile()

putLeoOutline(*p=None*)
Return a string, *not unicode*, encoded with self.leo_file_encoding, suitable for pasting to the clipboard.

putPostlog()

putPrefs()

putProlog()
Put the prolog of the xml file.

putReferencedTnodes()
Put all referenced tnodes.

putSavedMessage(*fileName*)

putStyleSheetLine()
Put the xml stylesheet line.

Leo 5.3: - Use only the stylesheet setting, ignoreing c.frame.stylesheet. - Write no stylesheet element if there is no setting.

The old way made it almost impossible to delete stylesheet element.

putTnode(*v*)

putTnodes()
Puts all tnodes as required for copy or save commands

putUaHelper(*torv*, *key*, *val*)
Put attribute whose name is key and value is val to the output stream.

putUnknownAttributes(*torv*)
Put pickleable values for all keys in torv.unknownAttributes dictionary.

putVnode(*p*, *isIgnore=False*)
Write a <v> element corresponding to a VNode.

putVnodes(*p=None*)
Puts all <v> elements in the order in which they appear in the outline.

putXMLLine()
Put the **properly encoded** <?xml> element.

put_dquote()

put_dquoted_bool(*b*)

put_flag(*a*, *b*)

put_in_dquotes(*a*)

put_nl()

put_tab()

put_tabs(*n*)

readAtFileNodes()

readExternalFiles(*fileName*)
Read all external files.

readOutlineOnly(*theFile*, *fileName*)

readSaxFile(*theFile*, *fileName*, *silent*, *inClipboard*, *reassignIndices*, *s=None*)
Read the entire .leo file using the sax parser.

reassignAllIndices(*p*)
Reassign all indices in p's subtree.

resolveArchivedPosition(*archivedPosition*, *root_v*)
Return a VNode corresponding to the archived position relative to root node root_v.

resolveTnodeLists()
Called *before* reading external files.

restoreDescendentAttributes()

retrieveVnodesFromDb(*conn*)
Recreates tree from the data contained in table vnodes. This method follows behavior of readSaxFile.

save(*fileName*, *silent=False*)

saveAs(*fileName*)

saveTo(*fileName*, *silent=False*)

save_ref()
Saves reference outline file

setDefaultDirectoryForNewFiles(*fileName*)
Set c.openDirectory for new files for the benefit of leoAtFile.scanAllDirectives.

setPositionsFromVnodes()

setReferenceFile(*fileName*)

updateFixedStatus()

updateFromRefFile()
Updates public part of outline from the specified file.

updateSaxClone(*sax_node*, *parent_v*, *v*)
Update the body text of v. It overrides any previous body text.

warnOnReadOnlyFiles(*fileName*)

writeAllAtFileNodesHelper()
Write all @<file> nodes and set orphan bits.

writeAtFileNodes(*event=None*)
Write all @file nodes in the selected outline.

writeAtShadowNodes(*event=None*)
Write all @file nodes in the selected outline.

writeDirtyAtFileNodes(*event=None*)
Write all changed @file Nodes.

writeDirtyAtShadowNodes(*event=None*)
Write all changed @shadow Nodes.

writeMissingAtFileNodes(*event=None*)
Write all @file nodes for which the corresponding external file does not exist.

writeOutlineOnly(*event=None*)
Write the entire outline without writing any derived files.

writeToFileHelper(*fileName*, *toOPML*)

writeToStringHelper(*fileName*)

writeZipFile(*s*)

write_LEO_file(*fileName*, *outlineOnlyFlag*, *toString=False*, *toOPML=False*)
Write the .leo file.

write_Leo_file(*fileName*, *outlineOnlyFlag*, *toString=False*, *toOPML=False*)
Write the .leo file.

exception leo.core.leoFileCommands.**InvalidPaste**
Bases: exceptions.Exception

class leo.core.leoFileCommands.**SaxContentHandler**(*c*, *fileName*, *silent*, *inClipboard*)
Bases: xml.sax.saxutils.XMLGenerator

A sax content handler class that reads Leo files.

attrsToList(*attrs*)
Convert the attributes to a list of g.Bunches.

attrs: an Attributes item passed to startElement.

attrsToString(*attrs*, *sep='\n'*)
Convert the attributes to a string.

attrs: an Attributes item passed to startElement.

sep: the separator charater between attributes.

characters(*content*)
Handle the characters element.

clean(*s*)

endDocument()

endElement(*name*)
Handle the end of any xml element.

endElementNS(*unused_name*, *unused_qname*)

endTnode()
Handle the end of a <tnode> element.

endVH()
Handle the end of a <vh> element.

endVnode()
Handle the end of a <vnode> element.

error(*message*)

getRootNode()

getWindowPositionAttributes(*attrs*)

ignorableWhitespace(*unused_whitespace*)

inElement(*name*)

printStartElement(*name*, *attrs*)

processingInstruction(*target*, *data*)
sax: handle an xml processing instruction. We expect the target to be 'xml-stylesheet'.

skippedEntity(*name*)

startDocument()

startElement(*name*, *attrs*)

startElementNS(*unused_name*, *unused_qname*, *unused_attrs*)

startGlobals(*attrs*)

startLeoHeader(*unused_attrs*)

startTnode(*attrs*)

startVH(*unused_attrs*)

startVnode(*attrs*)

startVnodes(*unused_attrs*)

startWinPos(*attrs*)

tnodeAttributes(*attrs*)

vnodeAttributes(*attrs*)

class leo.core.leoFileCommands.**SaxNodeClass**
A class representing one <v> element.

Use getters to access the attributes, properties and rules of this mode.

dump()

1.2.1.18 leoFind Module

Leo's gui-independent find classes.

class leo.core.leoFind.**LeoFind**(*c*)
Bases: object

The base class for Leo's Find commands.

abortSearch()
Restore the original position and selection.

addChangeStringToLabel()
Add an unprotected change string to the minibuffer label.

addFindStringToLabel(*protect=True*)

backwardsHelper(*s*, *i*, *j*, *pattern*, *nocase*, *word*)

batchChange(*pos1*, *pos2*)
Do a single batch change operation, updating the head or body string of p and leaving the result in s_ctrl.

s_ctrl contains the found text on entry and contains the changed text on exit. pos and pos2 indicate the selection. The selection will never be empty.

change(*event=None*)

changeAll()

changeAllButton(*event=None*)
Handle Replace All button.

changeAllCommand(*event=None*)

changeButton(*event=None*)
Handle Change button.

changeCommand(*event=None*)
Handle replace command.

changeSelection()

changeThenFind()

changeThenFindButton(*event=None*)
Handle Change, Then Find button.

changeThenFindCommand(*event=None*)
Handle the replace-then-find command.

checkArgs()

cloneFindAllCommand(*event=None*)

cloneFindAllFlattenedCommand(*event=None*)

cloneFindTag(*tag*)
Handle the clone-find-tag command.

cmd()
Command decorator for the findCommands class.

computeFindOptions()
Return the status line as two strings.

computeFindOptionsInStatusArea()

createCloneFindAllNodes(*clones*, *flattened*)
Create a "Found" node as the last node of the outline. Clone all positions in the clones set a children of found.

createCloneTagNodes(*clones*)
Create a "Found Tag" node as the last node of the outline. Clone all positions in the clones set as children of found.

createFindAllNode(*result*)
Create a "Found All" node as the last node of the outline.

createFindUniqueNode()
Create a "Found Unique" node as the last node of the outline.

debugIndices = []

doCloneFindAll(*after*, *data*, *flatten*, *p*, *undoType*)
Handle the clone-find-all command, from p to after.

doCloneFindAllHelper(*clones*, *count*, *flatten*, *p*, *skip*)
Handle the cff or cfa at node p.

doFindAll(*after*, *data*, *p*, *undoType*)
Handle the find-all command from p to after.

doWrap()
Return the position resulting from a wrap.

editWidget(*event*, *forceFocus=True*)
An override of baseEditCommands.editWidget that does *not* set focus when using anything other than the tk gui.

This prevents this class from caching an edit widget that is about to be deallocated.

endSearch()

escapeCommand(*event*)
Return the escaped command to execute.

findAll(*clone_find_all=False*, *clone_find_all_flattened=False*)

findAllButton(*event=None*)
Handle Find All button.

findAllCommand(*event=None*)

findButton(*event=None*)
Handle pressing the "Find" button in the find panel.

findDef(*event=None*)
Find the def or class under the cursor.

findDefHelper(*event*, *defFlag*)
Find the definition of the class, def or var under the cursor.

findEscapes()
Return the keystrokes corresponding to find-next & find-prev commands.

findNext(*initFlag=True*)
Find the next instance of the pattern.

findNextBatchMatch(*p*)
Find the next batch match at p.

findNextCommand(*event=None*)
The find-next command.

findNextMatch()
Resume the search where it left off.

findPrevCommand(*event=None*)
Handle F2 (find-previous)

findPreviousButton(*event=None*)
Handle the Find Previous button.

findVar(*event=None*)
Find the var under the cursor.

finishCreate()

firstSearchPane()
Set return the value of self.in_headline indicating which pane to search first.

focusInTree()
Return True is the focus widget w is anywhere in the tree pane.

Note: the focus may be in the find pane.

focusToFind(*event=None*)

generalChangeHelper(*find_pattern*, *change_pattern*, *changeAll=False*)

generalSearchHelper(*pattern*, *cloneFindAll=False*, *cloneFindAllFlattened=False*, *findAll=False*)

getFindResultStatus(*find_all=False*)
Return the status to be shown in the status line after a find command completes.

getStrokes(*commandName*)

helpForFindCommands(*event=None*)
Called from Find panel. Redirect.

hideFindTab(*event=None*)
Hide the Find tab.

iSearch(*again=False*)
Handle the actual incremental search.

iSearchBackspace()

iSearchStateHandler(*event*)
The state manager when the state is 'isearch

initBatchCommands()
Init for find-all and replace-all commands.

initBatchText(*ins=None*)
Init s_ctrl from self.p and ins at the beginning of a search.

initFindDef(*event*)
Init the find-def command. Return the word to find or None.

initInHeadline()
Select the first pane to search for incremental searches and changes. This is called only at the start of each search. This must not alter the current insertion point or selection range.

initInteractiveCommands()
Init an interactive command. This is tricky!

Always start in the presently selected widget, provided that searching is enabled for that widget. Always start at the present insert point for the body pane. For headlines, start at beginning or end of the headline text.

initNextText(*ins=None*)
Init s_ctrl when a search fails. On entry: - self.in_headline indicates what text to use. - self.reverse indicates how to set the insertion point.

init_s_ctrl(*s*, *ins*)
Init the contents of s_ctrl from s and ins.

isearchBackward(*event*)
Begin a backward incremental search.

- Plain characters extend the search backward.
- !<isearch-forward>! repeats the search.
- Esc or any non-plain key ends the search.
- Backspace reverses the search.
- Backspacing to an empty search pattern completely undoes the effect of the search.

isearchBackwardRegexp(*event*)
Begin a backward incremental regexp search.

- Plain characters extend the search.
- !<isearch-forward-regexp>! repeats the search.
- Esc or any non-plain key ends the search.
- Backspace reverses the search.
- Backspacing to an empty search pattern completely undoes the effect of the search.

isearchForward(*event*)
Begin a forward incremental search.

- Plain characters extend the search.
- !<isearch-forward>! repeats the search.
- Esc or any non-plain key ends the search.
- Backspace reverses the search.
- Backspacing to an empty search pattern completely undoes the effect of the search.

isearchForwardRegexp(*event*)
Begin a forward incremental regexp search.

- Plain characters extend the search.
- !<isearch-forward-regexp>! repeats the search.
- Esc or any non-plain key ends the search.
- Backspace reverses the search.
- Backspacing to an empty search pattern completely undoes the effect of the search.

isearchWithPresentOptions(*event*)
Begin an incremental search using find panel options.

- Plain characters extend the search.
- !<isearch-forward-regexp>! repeats the search.
- Esc or any non-plain key ends the search.
- Backspace reverses the search.

- Backspacing to an empty search pattern completely undoes the effect of the search.

lastStateHelper()

makeRegexSubs(*s*, *groups*)
Carefully substitute group[i-1] for i strings in s. The group strings may contain i strings: they are *not* substituted.

matchWord(*s*, *i*, *pattern*)
Do a whole-word search.

minibufferCloneFindAll(*event=None*, *preloaded=None*)
clone-find-all (aka find-clone-all and cfa).

Create an organizer node whose descendants contain clones of all nodes matching the search string, except @nosearch trees.

The list is *not* flattened: clones appear only once in the descendants of the organizer node.

minibufferCloneFindAll1(*event*)

minibufferCloneFindAllFlattened(*event=None*, *preloaded=None*)
clone-find-all-flattened (aka find-clone-all-flattened and cff).

Create an organizer node whose direct children are clones of all nodes matching the search string, except @nosearch trees.

The list is flattened: every cloned node appears as a direct child of the organizer node, even if the clone also is a descendant of another cloned node.

minibufferCloneFindAllFlattened1(*event*)

minibufferCloneFindTag(*event=None*)
clone-find-tag (aka find-clone-tag and cft).

Create an organizer node whose descendants contain clones of all nodes matching the given tag, except @nosearch trees.

The list is *always* flattened: every cloned node appears as a direct child of the organizer node, even if the clone also is a descendant of another cloned node.

minibufferCloneFindTag1(*event*)

minibufferFindAll(*event=None*)
Create a summary node containing descriptions of all matches of the search string.

minibufferFindAllUniqueRegex(*event=None*)
Create a summary node containing all unique matches of the regex search string. This command shows only the matched string itself.

minibufferReplaceAll(*event=None*)
Replace all instances of the search string with the replacement string.

minibufferTagChildren(*event=None*)
tag-children: prompt for a tag and add it to all children of c.p.

minibufferTagChildren1(*event*)

nextNodeAfterFail(*p*)
Return the next node after a failed search or None.

openFindTab(*event=None*, *show=True*)
Open the Find tab in the log pane.

outsideSearchRange(*p*)
Return True if the search is about to go outside its range, assuming both the headline and body text of the present node have been searched.

passedWrapPoint(*p*, *pos*, *newpos*)
Return True if the search has gone beyond the wrap point.

plainHelper(*s*, *i*, *j*, *pattern*, *nocase*, *word*)
Do a plain search.

pop()

precompilePattern()
Precompile the regexp pattern if necessary.

preloadFindPattern(*w*)
Preload the find pattern from the selected text of widget w.

printLine(*line*, *allFlag=False*)

push(*p*, *i*, *j*, *in_headline*)

reSearch1(*event*)

reSearchBackward(*event*)

reSearchForward(*event*)

regexHelper(*s*, *i*, *j*, *pattern*, *backwards*, *nocase*)

reloadSettings()
LeoFind.reloadSettings.

replace(*event=None*)

replaceBackSlashes(*s*)
Carefully replace backslashes in a search pattern.

resetWrap(*event=None*)

reset_state_ivars()
Reset ivars related to suboutline-only and wrapped searches.

restore(*data*)
Restore the screen and clear state after a search fails.

restoreAfterFindDef()
Restore find settings in effect before a find-def command.

restoreAllExpansionStates(*expanded*, *redraw=False*)
expanded is a set of gnx of nodes that should be expanded

returnToOrigin(*event*)

save()
Save everything needed to restore after a search fails.

saveBeforeFindDef(*p*)
Save the find settings in effect before a find-def command.

search()
Search s_ctrl for self.find_text with present options. Returns (pos, newpos) or (None,None).

search1(*event*)

searchBackward(*event*)

searchForward(*event*)

searchHelper(*s*, *i*, *j*, *pattern*)
Dispatch the proper search method based on settings.

searchWithPresentOptions(*event*, *findAllFlag=False*, *findAllUniqueFlag=False*, *changeAllFlag=False*)
Open the search pane and get the search string.

searchWithPresentOptions1(*event*)

setFindDefOptions(*p*)
Set the find options needed for the find-def command.

setFindScope(*where*)
Set the radio buttons to the given scope

setFindScopeEveryWhere(*event=None*)
Set the 'Entire Outline' radio button in the Find tab.

setFindScopeNodeOnly(*event=None*)
Set the 'Node Only' radio button in the Find tab.

setFindScopeSuboutlineOnly(*event=None*)
Set the 'Suboutline Only' radio button in the Find tab.

setReplaceString(*event*)
A state handler to get the replacement string.

setReplaceString1(*event*)

setReplaceString2(*event*)

setWidget()

setupArgs(*forward=False*, *regexp=False*, *word=False*)
Set up args for commands that force various values for commands (re-/word-/search-backward/forward) that force one or more of these values to be a spefic value.

setupChangePattern(*pattern*)

setupSearchPattern(*pattern*)

setup_button()
Init a search started by a button in the Find panel.

setup_command()

shouldStayInNode(*p*)
Return True if the find should simply switch panes.

showFindOptions(*event=None*)
Show the present find options in the status line. This is useful for commands like search-forward that do not show the Find Panel.

showFindOptionsInStatusArea()
Show find options in the status area.

showStatus(*found*)
Show the find status the Find dialog, if present, and the status line.

showSuccess(*pos*, *newpos*, *showState=True*)
Display the result of a successful find operation.

startIncremental(*event*, *commandName*, *forward*, *ignoreCase*, *regexp*)

startSearch(*event*)

stateZeroHelper(*event*, *prefix*, *handler*, *escapes=None*)

switchStyle(*word*)
Switch between camelCase and underscore_style function defintiions. Return None if there would be no change.

tagChildren(*tag*)
Handle the clone-find-tag command.

toggleFindCollapesNodes(*event*)
Toggle the 'Collapse Nodes' checkbox in the find tab.

toggleIgnoreCaseOption(*event*)
Toggle the 'Ignore Case' checkbox in the Find tab.

toggleMarkChangesOption(*event*)
Toggle the 'Mark Changes' checkbox in the Find tab.

toggleMarkFindsOption(*event*)
Toggle the 'Mark Finds' checkbox in the Find tab.

toggleOption(*checkbox_name*)

toggleRegexOption(*event*)
Toggle the 'Regexp' checkbox in the Find tab.

toggleSearchBodyOption(*event*)
Set the 'Search Body' checkbox in the Find tab.

toggleSearchHeadlineOption(*event*)
Toggle the 'Search Headline' checkbox in the Find tab.

toggleWholeWordOption(*event*)
Toggle the 'Whole Word' checkbox in the Find tab.

toggleWrapSearchOption(*event*)
Toggle the 'Wrap Around' checkbox in the Find tab.

updateChangeList(*s*)

updateFindList(*s*)

update_ivars()
Update ivars from the find panel.

wordSearch1(*event*)

wordSearchBackward(*event*)

wordSearchForward(*event*)

class leo.core.leoFind.**SearchWidget**(**args*, ***keys*)
Bases: object

A class to simulating high-level interface widget.

delete(*i*, *j=None*)

getAllText()

getInsertPoint()

getSelectionRange()

insert(*i*, *s*)

setAllText(*s*)

setInsertPoint(*i*, *s=None*)

setSelectionRange(*i*, *j*, *insert=None*)

toPythonIndex(*i*)

1.2.1.19 leoFrame Module

1.2.1.20 leoGlobals Module

Global constants, variables and utility functions used throughout Leo.

Important: This module imports no other Leo module.

class leo.core.leoGlobals.**BindingInfo**(*kind*, *commandName=''*, *func=None*, *nextMode=None*, *pane=None*, *stroke=None*)

Bases: object

A class representing any kind of key binding line.

This includes other information besides just the KeyStroke.

dump()

isModeBinding()

class leo.core.leoGlobals.**Bunch**(***keywords*)

Bases: object

A class that represents a colection of things.

Especially useful for representing a collection of related variables.

get(*key*, *theDefault=None*)

ivars()

keys()

toString()

leo.core.leoGlobals.**CheckVersion**(*s1*, *s2*, *condition='>='*, *stringCompare=None*, *delimiter='.'*, *trace=False*)

leo.core.leoGlobals.**CheckVersionToInt**(*s*)

class leo.core.leoGlobals.**Command**(*name*, ***kwargs*)

Bases: object

A global decorator for creating commands.

This is the recommended way of defining all new commands, including commands that could befined inside a class. The typical usage is:

@g.command('command-name') def A_Command(event):

c = event.get('c') ...

g can *not* be used anywhere in this class!

class leo.core.leoGlobals.**CommanderCommand**(*name*, ***kwargs*)

Bases: object

A global decorator for creating commander commands, that is, commands that were formerly methods of the Commands class in leoCommands.py.

Usage:

@g.command('command-name') def command_name(self, *args, **kwargs):

...

The decorator injects command_name into the Commander class and calls funcToMethod so the ivar will be injected in all future commanders.

g can *not* be used anywhere in this class!

class leo.core.leoGlobals.**FileLikeObject**(*encoding='utf-8'*, *fromString=None*)
Bases: object

Define a file-like object for redirecting writes to a string.

The caller is responsible for handling newlines correctly.

clear()

close()

flush()

get()

getvalue()

read()

readline()
Read the next line using at.list and at.ptr.

write(*s*)

class leo.core.leoGlobals.**GeneralSetting**(*kind*, *encoding=None*, *ivar=None*, *setting=None*, *val=None*, *path=None*, *tag='setting'*, *unl=None*)
Bases: object

A class representing any kind of setting except shortcuts.

dump()
x.__repr__() <==> repr(x)

class leo.core.leoGlobals.**GitIssueController**
Bases: object

A class encapsulating the retrieval of GitHub issues.

The GitHub api: https://developer.github.com/v3/issues/

backup_issues(*base_url*, *c*, *label_list*, *root*, *state=None*)

get_all_issues(*label_list*, *root*, *state*, *limit=100*)
Get all issues for the base url.

get_issues(*base_url*, *label_list*, *milestone*, *root*, *state*)
Create a list of issues for each label in label_list.

get_one_issue(*label*, *state*, *limit=20*)
Create a list of issues with the given label.

get_one_page(*label*, *page*, *r*, *root*)

print_header(*r*)

leo.core.leoGlobals.**IdleTime**(*handler*, *delay=500*, *tag=None*)

A thin wrapper for the LeoQtGui.IdleTime class.

The IdleTime class executes a handler with a given delay at idle time. The handler takes a single argument, the IdleTime instance:

```
def handler(timer):
    """IdleTime handler.  timer is an IdleTime instance."""
    delta_t = timer.time-timer.starting_time
    g.trace(timer.count, '%2.4f' % (delta_t))
    if timer.count >= 5:
        g.trace('done')
        timer.stop()

# Execute handler every 500 msec. at idle time.
timer = g.IdleTime(handler,delay=500)
if timer: timer.start()
```

Timer instances are completely independent:

```
def handler1(timer):
    delta_t = timer.time-timer.starting_time
    g.trace('%2s %2.4f' % (timer.count,delta_t))
    if timer.count >= 5:
        g.trace('done')
        timer.stop()

def handler2(timer):
    delta_t = timer.time-timer.starting_time
    g.trace('%2s %2.4f' % (timer.count,delta_t))
    if timer.count >= 10:
        g.trace('done')
        timer.stop()

timer1 = g.IdleTime(handler1,delay=500)
timer2 = g.IdleTime(handler2,delay=1000)
if timer1 and timer2:
    timer1.start()
    timer2.start()
```

class leo.core.leoGlobals.**KeyStroke**(*binding*)

Bases: object

A class that represent any key stroke or binding.

stroke.s is the "canonicalized" stroke.

dump()

Show results of printable chars.

finalize_binding(*binding*)

finalize_char(*s*)

Perform very-last-minute translations on bindings.

find(*pattern*)

find_mods(*s*)

Return the list of all modifiers seen in s.

isAltCtrl()
Return True if this is an Alt-Ctrl character.

isFKey()

isNumPadKey()

isPlainKey()
Return True if self.s represents a plain key.

A plain key is a key that can be inserted into text.

Note: The caller is responsible for handling Alt-Ctrl keys.

isPlainNumPad()

lower()

prettyPrint()

removeNumPadModifier()

startswith(*s*)

strip_mods(*s*)
Remove all modifiers from s, without changing the case of s.

strip_shift(*s*)
Handle supposedly shifted keys.

User settings might specify an already-shifted key, which is not an error.

The legacy Tk binding names have already been translated, so we don't have to worry about Shift-ampersand, etc.

toGuiChar()
Replace special chars by the actual gui char.

toInsertableChar()
Convert self to an (insertable) char.

class leo.core.leoGlobals.**MatchBrackets**(*c*, *p*, *language*)
Bases: object

A class implementing the match-brackets command. In the interest of speed, the code assumes that the user invokes the match-bracket command ouside of any string, comment or (for perl or javascript) regex.

back_scan_comment(*s*, *i*)
Return the index of the character after a comment.

ends_comment(*s*, *i*)
Return True if s[i] ends a comment. This is called while scanning backward, so this is a bit of a guess.

expand_range(*s*, *left*, *right*, *max_right*, *expand=False*)
Find the bracket nearest the cursor searching outwards left and right.

Expand the range (left, right) in string s until either s[left] or s[right] is a bracket. right can not exceed max_right, and if expand is True, the new range must encompass the old range, in addition to s[left] or s[right] being a bracket.

Returns new_left, new_right, bracket_char, index_of_bracket_char

if expansion succeeds, otherwise None, None, None, None

Note that only one of new_left and new_right will necessarily be a bracket, but index_of_bracket_char will definitely be a bracket.

find_matching_bracket(*ch1*, *s*, *i*)
Find the bracket matching s[i] for self.language.

is_regex(*s*, *i*)
Return true if there is another slash on the line.

oops(*s*)
Report an error in the match-brackets command.

run()
The driver for the MatchBrackets class.

With no selected range, find the nearest bracket and select from it to it's match, moving cursor to mathc. With selected range, the first time, move cursor back to other end of range. The second time, select enclosing range.

scan(*ch1*, *target*, *s*, *i*)
Scan forward for target.

scan_back(*ch1*, *target*, *s*, *i*)
Scan backwards for delim.

scan_comment(*s*, *i*)
Return the index of the character after a comment.

scan_regex(*s*, *i*)
Scan a regex (or regex substitution for perl).

scan_string(*s*, *i*)
Scan the string starting at s[i] (forward or backward). Return the index of the next character.

starts_comment(*s*, *i*)
Return True if s[i] starts a comment.

class leo.core.leoGlobals.**NullObject**(**args*, ***keys*)
Bases: object

An object that does nothing, and does it very well. From the Python cookbook, recipe 5.23

class leo.core.leoGlobals.**PosList**(*c*, *aList=None*)
Bases: list

A subclass of list for creating and selecting lists of positions.

This is deprecated, use leoNodes.PosList instead!

aList = g.PosList(c) # Creates a PosList containing all positions in c.

aList = g.PosList(c,aList2) # Creates a PosList from aList2.

aList2 = aList.select(pattern,regex=False,removeClones=True) # Creates a PosList containing all positions p in aList # such that p.h matches the pattern. # The pattern is a regular expression if regex is True. # if removeClones is True, all positions p2 are removed # if a position p is already in the list and p2.v == p.v.

aList.dump(sort=False,verbose=False) # Prints all positions in aList, sorted if sort is True. # Prints p.h, or repr(p) if verbose is True.

dump(*sort=False*, *verbose=False*)

removeClones(*aList*)

select(*pat*, *regex=False*, *removeClones=True*)
Return a new PosList containing all positions in self that match the given pattern.

class leo.core.leoGlobals.**ReadLinesClass**(*s*)
Bases: object

A class whose next method provides a readline method for Python's tokenize module.

next()

class leo.core.leoGlobals.**RedirectClass**
Bases: object

A class to redirect stdout and stderr to Leo's log pane.

flush(**args*)

isRedirected()

rawPrint(*s*)

redirect(*stdout=1*)

undirect(*stdout=1*)

write(*s*)

class leo.core.leoGlobals.**SherlockTracer**(*patterns*, *dots=True*, *show_args=True*, *show_return=True*, *verbose=True*)
Bases: object

A stand-alone tracer class with many of Sherlock's features.

This class should work in any environment containing the re, os and sys modules.

The arguments in the pattern lists determine which functions get traced or which stats get printed. Each pattern starts with "+", "-", "+:" or "-:", followed by a regular expression:

```
"+x"  Enables tracing (or stats) for all functions/methods whose name
```

matches the regular expression x.

"-x" Disables tracing for functions/methods. "+:x" Enables tracing for all functions in the **file** whose name matches x. "-:x" Disables tracing for an entire file.

Enabling and disabling depends on the order of arguments in the pattern list. Consider the arguments for the Rope trace:

patterns=['+.*','+:.*', '-:.*lib.*','+:.*rope.*','-:.*leoGlobals.py', '-:.*worder.py','-:.*prefs.py','-:.*resources.py',])

This enables tracing for everything, then disables tracing for all library modules, except for all rope modules. Finally, it disables the tracing for Rope's worder, prefs and resources modules. Btw, this is one of the best uses for regular expressions that I know of.

Being able to zero in on the code of interest can be a big help in studying other people's code. This is a non-invasive method: no tracing code needs to be inserted anywhere.

bad_pattern(*pattern*)
Report a bad Sherlock pattern.

check_pattern(*pattern*)
Give an error and return False for an invalid pattern.

dispatch(*frame*, *event*, *arg*)
The dispatch method.

do_call(*frame*, *unused_arg*)
Trace through a function call.

do_line(*frame*, *arg*)
print each line of enabled functions.

do_return(*frame*, *arg*)
Trace a return statement.

fn_is_enabled(*fn*, *patterns*)
Return True if tracing for fn is enabled. Used only to enable *statistics* for fn.

format_ret(*arg*)
Format arg, the value returned by a "return" statement.

get_args(*frame*)
Return name=val for each arg in the function call.

get_full_name(*locals_*, *name*)
Return class_name::name if possible.

is_enabled(*fn*, *name*, *patterns=None*)
Return True if tracing for name in fn is enabled.

pop()
Restore the pushed patterns.

print_stats(*patterns=None*)
Print all accumulated statisitics.

push(*patterns*)
Push the old patterns and set the new.

run(*frame=None*)
Trace from the given frame or the caller's frame.

set_patterns(*patterns*)
Set the patterns in effect.

show(*item*)
return the best representation of item.

stop()
Stop all tracing.

class leo.core.leoGlobals.**Tracer**(*limit=0*, *trace=False*, *verbose=False*)
Bases: object

A "debugger" that computes a call graph.

To trace a function and its callers, put the following at the function's start:

g.startTracer()

computeName(*frame*)

report()

stop()

tracer(*frame*, *event*, *arg*)
A function to be passed to sys.settrace.

updateStats(*name*)

class leo.core.leoGlobals.**TypedDict**(*name*, *keyType*, *valType*)
Bases: object

A class containing a name and enforcing type checking.

add(*key*, *val*)

copy(*name=None*)
Return a new dict with the same contents.

dump()

get(*key*, *default=None*)

get_setting(*key*)

get_string_setting(*key*)

keys()

name()

replace(*key*, *val*)

setName(*name*)

update(*d*)

class leo.core.leoGlobals.**TypedDictOfLists**(*name*, *keyType*, *valType*)
Bases: *leo.core.leoGlobals.TypedDict*

A class whose values are lists of typed values.

copy(*name=None*)
Return a new dict with the same contents.

exception leo.core.leoGlobals.**UiTypeException**
Bases: exceptions.Exception

leo.core.leoGlobals.**act_on_node**(*c*, *p*, *event*)

leo.core.leoGlobals.**actualColor**(*color*)
Return the actual color corresponding to the requested color.

leo.core.leoGlobals.**adjustTripleString**(*s*, *tab_width*)
Remove leading indentation from a triple-quoted string.

This works around the fact that Leo nodes can't represent underindented strings.

leo.core.leoGlobals.**alert**(*message*, *c=None*)
Raise an alert.

This method is deprecated: use c.alert instead.

leo.core.leoGlobals.**angleBrackets**(*s*)

leo.core.leoGlobals.**assertUi**(*uitype*)

leo.core.leoGlobals.**assert_is**(*obj*, *list_or_class*, *warn=True*)

leo.core.leoGlobals.**backupGitIssues**(*c*, *base_url=None*)
Get a list of issues from Leo's GitHub site.

leo.core.leoGlobals.**blue**(**args*, ***keys*)

leo.core.leoGlobals.**bunch**
alias of *leo.core.leoGlobals.Bunch*

leo.core.leoGlobals.**callback**(*func*)
A global decorator that protects Leo against crashes in callbacks.

This is the recommended way of defining all callback.

@g.callback def a_callback(...):

c = event.get('c') ...

leo.core.leoGlobals.**caller**(*i=1*)
Return the caller name i levels up the stack.

leo.core.leoGlobals.**callers**(*n=4*, *count=0*, *excludeCaller=True*, *verbose=False*)
Return a list containing the callers of the function that called g.callerList.

excludeCaller: True (the default), g.callers itself is not on the list.

If the *verbose* keyword is True, or the caller name is in the *names* list, return:

line N <file name> <class_name>.<caller_name> # methods line N <file name> <caller_name> # functions.

Otherwise, just return the <caller name>

leo.core.leoGlobals.**cantImport**(*moduleName*, *pluginName=None*, *verbose=True*)
Print a "Can't Import" message and return None.

leo.core.leoGlobals.**chdir**(*path*)

leo.core.leoGlobals.**checkOpenDirectory**(*c*)

leo.core.leoGlobals.**checkUnchangedIvars**(*obj*, *d*, *exceptions=None*)

leo.core.leoGlobals.**check_cmd_instance_dict**(*c*, *g*)
Check g.check_cmd_instance_dict. This is a permanent unit test, called from c.finishCreate.

leo.core.leoGlobals.**choose**(*cond*, *a*, *b*)
(Deprecated) simulate "a if cond else b"

leo.core.leoGlobals.**clearAllIvars**(*o*)
Clear all ivars of o, a member of some class.

leo.core.leoGlobals.**clearStats**()

leo.core.leoGlobals.**cls**(*event=None*)
Clear the screen.

leo.core.leoGlobals.**collectGarbage**()

leo.core.leoGlobals.**command**
alias of *leo.core.leoGlobals.Command*

leo.core.leoGlobals.**command_alias**(*alias*, *func*)
Create an alias for the *already defined* method in the Commands class.

leo.core.leoGlobals.**commander_command**
alias of *leo.core.leoGlobals.CommanderCommand*

leo.core.leoGlobals.**comment_delims_from_extension**(*filename*)
Return the comment delims corresponding to the filename's extension.

```
>>> import leo.core.leoGlobals as g
>>> g.comment_delims_from_extension(".py")
('#', '', '')
```

```
>>> g.comment_delims_from_extension(".c")
('//', '/*', '*/')
```

```
>>> g.comment_delims_from_extension(".html")
('', '<!--', '-->')
```

leo.core.leoGlobals.**composeScript**(*c*, *p*, *s*, *forcePythonSentinels=True*, *useSentinels=True*)
Compose a script from p.b.

leo.core.leoGlobals.**computeBaseDir**(*c*, *base_dir*, *path_setting*, *trace=False*)
Compute a base_directory. If given, @string path_setting takes precedence.

leo.core.leoGlobals.**computeCommands**(*c*, *commands*, *command_setting*, *trace=False*)
Get the list of commands. If given, @data command_setting takes precedence.

leo.core.leoGlobals.**computeFileUrl**(*fn*, *c=None*, *p=None*)
Compute finalized url for filename fn. This involves adding url escapes and evaluating Leo expressions.

leo.core.leoGlobals.**computeGlobalConfigDir**()

leo.core.leoGlobals.**computeHomeDir**()

leo.core.leoGlobals.**computeLeadingWhitespace**(*width*, *tab_width*)

leo.core.leoGlobals.**computeLeadingWhitespaceWidth**(*s*, *tab_width*)

leo.core.leoGlobals.**computeLeoDir**()

leo.core.leoGlobals.**computeLoadDir**()

leo.core.leoGlobals.**computeMachineName**()

leo.core.leoGlobals.**computeStandardDirectories**()

leo.core.leoGlobals.**computeWidth**(*s*, *tab_width*)

leo.core.leoGlobals.**computeWindowTitle**(*fileName*)

leo.core.leoGlobals.**compute_directives_re**()
Return an re pattern which word matches all Leo directives. Only g.get_directives_dict uses this pattern.

leo.core.leoGlobals.**convertPythonIndexToRowCol**(*s*, *i*)
Convert index i into string s into zero-based row/col indices.

leo.core.leoGlobals.**convertRowColToPythonIndex**(*s*, *row*, *col*, *lines=None*)
Convert zero-based row/col indices into a python index into string s.

leo.core.leoGlobals.**createScratchCommander**(*fileName=None*)

leo.core.leoGlobals.**createTopologyList**(*c*, *root=None*, *useHeadlines=False*)
Creates a list describing a node and all its descendents

leo.core.leoGlobals.**create_temp_file**(*textMode=False*)
Return a tuple (theFile,theFileName)

theFile: a file object open for writing. theFileName: the name of the temporary file.

leo.core.leoGlobals.**defaultLeoFileExtension**(*c=None*)

leo.core.leoGlobals.**dictToString**(*d*, *indent=''*, *tag=None*)
Pretty print a Python dict to a string.

leo.core.leoGlobals.**disableIdleTimeHook**()
Disable the global idle-time hook.

leo.core.leoGlobals.**doHook**(*tag*, **args*, ***keywords*)
This global function calls a hook routine. Hooks are identified by the tag param.

Returns the value returned by the hook routine, or None if the there is an exception.

We look for a hook routine in three places: 1. c.hookFunction 2. app.hookFunction 3. leoPlugins.doPlugins()

Set app.hookError on all exceptions. Scripts may reset app.hookError to try again.

leo.core.leoGlobals.**doKeywordArgs**(*keys*, *d=None*)
Return a result dict that is a copy of the keys dict with missing items replaced by defaults in d dict.

leo.core.leoGlobals.**dummy_act_on_node**(*c*, *p*, *event*)

leo.core.leoGlobals.**dump**(*s*)

leo.core.leoGlobals.**dump_encoded_string**(*encoding*, *s*)
Dump s, assumed to be an encoded string.

leo.core.leoGlobals.**ecnl**(*tabName='Log'*)

leo.core.leoGlobals.**ecnls**(*n*, *tabName='Log'*)

leo.core.leoGlobals.**enableIdleTimeHook**(**args*, ***keys*)
Enable idle-time processing.

leo.core.leoGlobals.**enable_gc_debug**(*event=None*)

leo.core.leoGlobals.**enl**(*tabName='Log'*)

leo.core.leoGlobals.**ensureLeadingNewlines**(*s*, *n*)

leo.core.leoGlobals.**ensureTrailingNewlines**(*s*, *n*)

leo.core.leoGlobals.**ensure_extension**(*name*, *ext*)

leo.core.leoGlobals.**error**(**args*, ***keys*)

leo.core.leoGlobals.**es**(**args*, ***keys*)
Put all non-keyword args to the log pane. The first, third, fifth, etc. arg translated by g.translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.core.leoGlobals.**esDiffTime**(*message*, *start*)

leo.core.leoGlobals.**es_debug**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane in orange.

The first, third, fifth, etc. arg translated by g.translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.core.leoGlobals.**es_dump**(*s*, *n=30*, *title=None*)

leo.core.leoGlobals.**es_error**(**args*, ***keys*)

leo.core.leoGlobals.**es_event_exception**(*eventName*, *full=False*)

leo.core.leoGlobals.**es_exception**(*full=True*, *c=None*, *color='red'*)

leo.core.leoGlobals.**es_exception_type**(*c=None*, *color='red'*)

leo.core.leoGlobals.**es_print**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane.

The first, third, fifth, etc. arg translated by g.translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.core.leoGlobals.**es_print_error**(**args*, ***keys*)

leo.core.leoGlobals.**es_print_exception**(*full=True*, *c=None*, *color='red'*)
Print exception info about the last exception.

leo.core.leoGlobals.**es_trace**(**args*, ***keys*)

leo.core.leoGlobals.**escaped**(*s*, *i*)

leo.core.leoGlobals.**execGitCommand**(*command*, *directory=None*)
Execute the given git command in the given directory.

leo.core.leoGlobals.**exec_file**(*path*, *d*, *script=None*)
Simulate python's execfile statement for python 3.

leo.core.leoGlobals.**executeFile**(*filename*, *options=''*)

leo.core.leoGlobals.**executeScript**(*name*)
Execute a script whose short python file name is given.

This is called only from the scripts_menu plugin.

leo.core.leoGlobals.**execute_shell_commands**(*commands*, *trace=False*)
Execute each shell command in a separate process. Wait for each command to complete, except those starting with '&'

leo.core.leoGlobals.**execute_shell_commands_with_options**(*base_dir=None*, *c=None*, *command_setting=None*, *commands=None*, *path_setting=None*, *trace=False*, *warning=None*)
A helper for prototype commands or any other code that runs programs in a separate process.

base_dir: Base directory to use if no config path given. commands: A list of commands, for g.execute_shell_commands. commands_setting: Name of @data setting for commands. path_setting: Name of @string setting for the base directory. warning: A warning to be printed before executing the commands.

leo.core.leoGlobals.**extractExecutableString**(*c*, *p*, *s*, *language='python'*)
Return all lines for the given @language directive.

Ignore all lines under control of any other @language directive.

leo.core.leoGlobals.**fileFilters**(*key*)

leo.core.leoGlobals.**fileLikeObject**
alias of *leo.core.leoGlobals.FileLikeObject*

leo.core.leoGlobals.**file_date**(*theFile*, *format=None*)

leo.core.leoGlobals.**findLanguageDirectives**(*c*, *p*)
Return the language in effect at position p.

leo.core.leoGlobals.**findNodeAnywhere**(*c*, *headline*, *exact=True*)

leo.core.leoGlobals.**findNodeInChildren**(*c*, *p*, *headline*, *exact=True*)
Search for a node in v's tree matching the given headline.

leo.core.leoGlobals.**findNodeInTree**(*c*, *p*, *headline*, *exact=True*)
Search for a node in v's tree matching the given headline.

leo.core.leoGlobals.**findReference**(*name*, *root*)
Find the section definition for name.

If a search of the descendants fails, and an ancestor is an @root node, search all the descendants of the @root node.

leo.core.leoGlobals.**findRootsWithPredicate**(*c*, *root*, *predicate=None*)
Commands often want to find one or more **roots**, given a position p. A root is the position of any node matching a predicate.

This function formalizes the search order used by the pylint, pyflakes and the rst3 commands, returning a list of zero or more found roots.

leo.core.leoGlobals.**findTabWidthDirectives**(*c*, *p*)
Return the language in effect at position p.

leo.core.leoGlobals.**findTestScript**(*c*, *h*, *where=None*, *warn=True*)

leo.core.leoGlobals.**findTopLevelNode**(*c*, *headline*, *exact=True*)

leo.core.leoGlobals.**find_line_start**(*s*, *i*)
Return the index in s of the start of the line containing s[i].

leo.core.leoGlobals.**find_on_line**(*s*, *i*, *pattern*)

leo.core.leoGlobals.**find_word**(*s*, *word*, *i=0*)
Return the index of the first occurance of word in s, or -1 if not found.

g.find_word is *not* the same as s.find(i,word); g.find_word ensures that only word-matches are reported.

leo.core.leoGlobals.**flatten_list**(*obj*)
A generator yielding a flattened (concatenated) version of obj.

leo.core.leoGlobals.**fullPath**(*c*, *p*, *simulate=False*)
Return the full path (including fileName) in effect at p. Neither the path nor the fileName will be created if it does not exist.

leo.core.leoGlobals.**funcToMethod**(*f*, *theClass*, *name=None*)
From the Python Cookbook...

The following method allows you to add a function as a method of any class. That is, it converts the function to a method of the class. The method just added is available instantly to all existing instances of the class, and to all instances created in the future.

The function's first argument should be self.

The newly created method has the same name as the function unless the optional name argument is supplied, in which case that name is used as the method name.

leo.core.leoGlobals.**getBaseDirectory**(*c*)
Convert '!' or '.' to proper directory references.

leo.core.leoGlobals.**getDocString**(*s*)
Return the text of the first docstring found in s.

leo.core.leoGlobals.**getDocStringForFunction**(*func*)
Return the docstring for a function that creates a Leo command.

leo.core.leoGlobals.**getEncodingAt**(*p*, *s=None*)
Return the encoding in effect at p and/or for string s.

Read logic: s is not None. Write logic: s is None.

leo.core.leoGlobals.**getGitIssues**(*c*, *base_url=None*, *label_list=None*, *milestone=None*, *state=None*)
Get a list of issues from Leo's GitHub site.

leo.core.leoGlobals.**getHandlersForTag**(*tags*)

leo.core.leoGlobals.**getIvarsDict**(*obj*)
Return a dictionary of ivars:values for non-methods of obj.

leo.core.leoGlobals.**getLanguageAtPosition**(*c*, *p*)
Return the language in effect at position p. This is always a lowercase language name, never None.

leo.core.leoGlobals.**getLanguageFromAncestorAtFileNode**(*p*)
Return the language in effect as determined by the file extension of the nearest enclosing @<file> node.

leo.core.leoGlobals.**getLastTracebackFileAndLineNumber**()

leo.core.leoGlobals.**getLine**(*s*, *i*)
Return i,j such that s[i:j] is the line surrounding s[i]. s[i] is a newline only if the line is empty. s[j] is a newline unless there is no trailing newline.

leo.core.leoGlobals.**getLineAfter**(*s*, *i*)

leo.core.leoGlobals.**getLoadedPlugins**()

leo.core.leoGlobals.**getOutputNewline**(*c=None*, *name=None*)
Convert the name of a line ending to the line ending itself.

Priority: - Use name if name given - Use c.config.output_newline if c given, - Otherwise use g.app.config.output_newline.

leo.core.leoGlobals.**getPluginModule**(*moduleName*)

leo.core.leoGlobals.**getPythonEncodingFromString**(*s*)
Return the encoding given by Python's encoding line. s is the entire file.

leo.core.leoGlobals.**getScript**(*c*, *p*, *useSelectedText=True*, *forcePythonSentinels=True*, *useSentinels=True*)
Return the expansion of the selected text of node p. Return the expansion of all of node p's body text if p is not the current node or if there is no text selection.

leo.core.leoGlobals.**getTestVars**()

leo.core.leoGlobals.**getTime**()

leo.core.leoGlobals.**getUrlFromNode**(*p*)
Get an url from node p: 1. Use the headline if it contains a valid url. 2. Otherwise, look *only* at the first line of the body.

leo.core.leoGlobals.**getWord**(*s*, *i*)
Return i,j such that s[i:j] is the word surrounding s[i].

leo.core.leoGlobals.**get_directives_dict**(*p*, *root=None*)
Scan p for @directives found in globalDirectiveList.

Returns a dict containing the stripped remainder of the line following the first occurrence of each recognized directive

leo.core.leoGlobals.**get_directives_dict_list**(*p*)
Scans p and all its ancestors for directives.

Returns a list of dicts containing pointers to the start of each directive

leo.core.leoGlobals.**get_leading_ws**(*s*)
Returns the leading whitespace of 's'.

leo.core.leoGlobals.**get_line**(*s*, *i*)

leo.core.leoGlobals.**get_line_after**(*s*, *i*)

leo.core.leoGlobals.**gitBranchName**(*path=None*)
Return the git branch name associated with path/.git, or the empty string if path/.git does not exist. If path is None, use the leo-editor directory.

leo.core.leoGlobals.**gitCommitNumber**(*path=None*)
Return the git commit number associated with path/.git, or the empty string if path/.git does not exist. If path is None, use the leo-editor directory.

leo.core.leoGlobals.**gitDescribe**(*path=None*)

Return the Git tag, distance-from-tag, and commit hash for the associated path. If path is None, use the leo-editor directory.

Given *git describe* cmd line output: ‘x-leo-v5.6-55-ge1129da

‘ This function returns (‘x-leo-v5.6’, ‘55’, ‘e1129da’)

leo.core.leoGlobals.**gitHeadPath**(*path=None*)
Compute the path to the .git/HEAD directory given the path to another directory. If no path is given, use the path to *this* file. This code can *not* use g.app.loadDir because it is called too early in Leo’s startup code.

leo.core.leoGlobals.**gitInfo**(*path=None*)
Path is a .git/HEAD directory, or None.

Return the branch and commit number or (‘’, ‘’).

leo.core.leoGlobals.**glob_glob**(*pattern*)
Return the regularized glob.glob(pattern)

leo.core.leoGlobals.**goto_last_exception**(*c*)
Go to the line given by sys.last_traceback.

leo.core.leoGlobals.**guessExternalEditor**(*c=None*)
Return a ‘sensible’ external editor

leo.core.leoGlobals.**handleScriptException**(*c*, *p*, *script*, *script1*)

leo.core.leoGlobals.**handleUnl**(*unl*, *c*)
Handle a Leo UNL. This must *never* open a browser.

leo.core.leoGlobals.**handleUrl**(*url*, *c=None*, *p=None*)
Open a url or a unl.

leo.core.leoGlobals.**handleUrlHelper**(*url*, *c*, *p*)
Open a url. Most browsers should handle: ftp://ftp.uu.net/public/whatever http://localhost/MySiteUnderDevelopment/index.html file:///home/me/todolist.html

leo.core.leoGlobals.**idleTimeHookHandler**(*timer*)
This function exists for compatibility.

leo.core.leoGlobals.**importExtension**(*moduleName*, *pluginName=None*, *verbose=False*, *required=False*)
Try to import a module. If that fails, try to import the module from Leo’s extensions directory.

moduleName is the module’s name, without file extension.

leo.core.leoGlobals.**importFromPath**(*moduleName*, *path*, *verbose=False*)
Import a module whose name is given from the directory given by path.

Warning: This is a thin wrapper for imp.load_module, which is equivalent to reload! Reloading Leo files while running will crash Leo.

leo.core.leoGlobals.**importModule**(*moduleName*, *pluginName=None*, *verbose=False*)
Try to import a module as Python’s import command does.

moduleName is the module’s name, without file extension.

This function first attempts to import from sys.modules, then from the extensions and external directories.

leo.core.leoGlobals.**initScriptFind**(*c*, *findHeadline*, *changeHeadline=None*, *firstNode=None*, *script_search=True*, *script_change=True*)

leo.core.leoGlobals.**init_dialog_folder**(*c*, *p*, *use_at_path=True*)
Return the most convenient folder to open or save a file.

leo.core.leoGlobals.**init_zodb**(*pathToZodbStorage*, *verbose=True*)
Return an ZODB.DB instance from ZODB.FileStorage.FileStorage(pathToZodbStorage) return None on any error.

leo.core.leoGlobals.**input_**(*message=''*, *c=None*)
Safely execute python's input statement.

c.executeScriptHelper binds 'input' to be a wrapper that calls **g.input_** with c and handler bound properly.

leo.core.leoGlobals.**insertCodingLine**(*encoding*, *script*)
Insert a coding line at the start of script s if no such line exists. The coding line must start with @first because it will be passed to at.writeFromString.

leo.core.leoGlobals.**internalError**(**args*)
Report a serious interal error in Leo.

leo.core.leoGlobals.**isBindingInfo**(*obj*)

leo.core.leoGlobals.**isBytes**(*s*)
Return True if s is Python3k bytes type.

leo.core.leoGlobals.**isCallable**(*obj*)

leo.core.leoGlobals.**isDirective**(*s*)
Return True if s starts with a directive.

leo.core.leoGlobals.**isGeneralSetting**(*obj*)

leo.core.leoGlobals.**isInt**(*obj*)
Return True if obj is an int or a long.

leo.core.leoGlobals.**isList**(*s*)
Return True if s is a list.

leo.core.leoGlobals.**isMacOS**()

leo.core.leoGlobals.**isString**(*s*)
Return True if s is any string, but not bytes.

leo.core.leoGlobals.**isStroke**(*obj*)

leo.core.leoGlobals.**isStrokeOrNone**(*obj*)

leo.core.leoGlobals.**isTextWidget**(*w*)

leo.core.leoGlobals.**isTextWrapper**(*w*)

leo.core.leoGlobals.**isTypedDict**(*obj*)

leo.core.leoGlobals.**isTypedDictOfLists**(*obj*)

leo.core.leoGlobals.**isUnicode**(*s*)
Return True if s is a unicode string.

leo.core.leoGlobals.**isValidEncoding**(*encoding*)
Return True if the encooding is valid.

leo.core.leoGlobals.**isValidUrl**(*url*)
Return true if url *looks* like a valid url.

leo.core.leoGlobals.**isWordChar**(*ch*)
Return True if ch should be considered a letter.

leo.core.leoGlobals.**isWordChar1**(*ch*)

leo.core.leoGlobals.**is_binary_external_file**(*fileName*)

leo.core.leoGlobals.**is_binary_file**(*f*)

leo.core.leoGlobals.**is_binary_string**(*s*)

leo.core.leoGlobals.**is_c_id**(*ch*)

leo.core.leoGlobals.**is_nl**(*s*, *i*)

leo.core.leoGlobals.**is_sentinel**(*line*, *delims*)
Return True if line starts with a sentinel comment.

```
>>> import leo.core.leoGlobals as g
>>> py_delims = g.comment_delims_from_extension('.py')
>>> g.is_sentinel("#@+node",py_delims)
True
>>> g.is_sentinel("#comment",py_delims)
False
```

```
>>> c_delims = g.comment_delims_from_extension('.c')
>>> g.is_sentinel("//@+node",c_delims)
True
>>> g.is_sentinel("//comment",c_delims)
False
```

```
>>> html_delims = g.comment_delims_from_extension('.html')
>>> g.is_sentinel("<!--@+node-->",html_delims)
True
>>> g.is_sentinel("<!--comment-->",html_delims)
False
```

leo.core.leoGlobals.**is_special**(*s*, *i*, *directive*)
Return True if the body text contains the @ directive.

leo.core.leoGlobals.**is_ws**(*c*)

leo.core.leoGlobals.**is_ws_or_nl**(*s*, *i*)

leo.core.leoGlobals.**itemsMatchingPrefixInList**(*s*, *aList*, *matchEmptyPrefix=False*)
This method returns a sorted list items of aList whose prefix is s.

It also returns the longest common prefix of all the matches.

leo.core.leoGlobals.**ivars2instance**(*c*, *g*, *ivars*)
Return the instance of c given by ivars. ivars is a list of strings. A special case: ivars may be 'g', indicating the leoGlobals module.

leo.core.leoGlobals.**joinLines**(*aList*)

leo.core.leoGlobals.**join_list**(*aList*, *indent=''*, *leading=''*, *sep=''*, *trailing=''*)
Create a dict representing the concatenation of the strings in aList, formatted per the keyword args. See the HTMLReportTraverser class for many examples.

leo.core.leoGlobals.**joinlines**(*aList*)

leo.core.leoGlobals.**jsonCommitInfo**()
return asctime and timestamp from leo/core/commit_timestamp.json. return (‘’, ‘’) if the file does not exist or is not a valid .json file.

leo.core.leoGlobals.**listToString**(*obj*, *indent=”*, *tag=None*)
Pretty print a Python list to a string.

leo.core.leoGlobals.**list_to_string**(*obj*)
Convert obj (a list of lists) to a single string.

This function stresses the gc; it will usually be better to work with the much smaller strings generated by flatten_list.

Use this function only in special circumstances, for example, when it is known that the resulting string will be small.

leo.core.leoGlobals.**loadOnePlugin**(*pluginName*, *verbose=False*)

leo.core.leoGlobals.**log**(**args*, ***keys*)
Put all non-keyword args to the log pane. The first, third, fifth, etc. arg translated by g.translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.core.leoGlobals.**log_to_file**(*s*, *fn=None*)
Write a message to ~/test/leo_log.txt.

leo.core.leoGlobals.**longestCommonPrefix**(*s1*, *s2*)
Find the longest prefix common to strings s1 and s2.

leo.core.leoGlobals.**makeAllNonExistentDirectories**(*theDir*, *c=None*, *force=False*, *verbose=True*)
Attempt to make all non-existent directories

leo.core.leoGlobals.**makeDict**(***keys*)
Returns a Python dictionary from using the optional keyword arguments.

leo.core.leoGlobals.**makePathRelativeTo**(*fullPath*, *basePath*)

leo.core.leoGlobals.**match**(*s*, *i*, *pattern*)

leo.core.leoGlobals.**match_c_word**(*s*, *i*, *name*)

leo.core.leoGlobals.**match_ignoring_case**(*s1*, *s2*)

leo.core.leoGlobals.**match_word**(*s*, *i*, *pattern*)

leo.core.leoGlobals.**module_date**(*mod*, *format=None*)

leo.core.leoGlobals.**new_cmd_decorator**(*name*, *ivars*)
Return a new decorator for a command with the given name. Compute the class *instance* using the ivar string or list.

leo.core.leoGlobals.**note**(**args*, ***keys*)

leo.core.leoGlobals.**nullObject**
alias of *leo.core.leoGlobals.NullObject*

leo.core.leoGlobals.**objToString**(*obj*, *indent=”*, *printCaller=False*, *tag=None*)
Pretty print any Python object to a string.

leo.core.leoGlobals.**oldDump**(*s*)

leo.core.leoGlobals.**openUrl**(*p*)
Open the url of node p. Use the headline if it contains a valid url. Otherwise, look *only* at the first line of the body.

leo.core.leoGlobals.**openUrlHelper**(*event*, *url=None*)
Open the UNL or URL under the cursor. Return it for unit testing.

leo.core.leoGlobals.**openUrlOnClick**(*event*, *url=None*)
Open the URL under the cursor. Return it for unit testing.

leo.core.leoGlobals.**openWithFileName**(*fileName*, *old_c=None*, *gui=None*)
Create a Leo Frame for the indicated fileName if the file exists.

returns the commander of the newly-opened outline.

leo.core.leoGlobals.**optimizeLeadingWhitespace**(*line*, *tab_width*)

leo.core.leoGlobals.**os_path_abspath**(*path*)
Convert a path to an absolute path.

leo.core.leoGlobals.**os_path_basename**(*path*)
Return the second half of the pair returned by split(path).

leo.core.leoGlobals.**os_path_dirname**(*path*)
Return the first half of the pair returned by split(path).

leo.core.leoGlobals.**os_path_exists**(*path*)
Return True if path exists.

leo.core.leoGlobals.**os_path_expandExpression**(*s*, ***keys*)
Expand all {{anExpression}} in c's context.

leo.core.leoGlobals.**os_path_expanduser**(*path*)
wrap os.path.expanduser

leo.core.leoGlobals.**os_path_finalize**(*path*, ***keys*)
Expand '~', then return os.path.normpath, os.path.abspath of the path. There is no corresponding os.path method

leo.core.leoGlobals.**os_path_finalize_join**(**args*, ***keys*)
Do os.path.join(*args), then finalize the result.

leo.core.leoGlobals.**os_path_getmtime**(*path*)
Return the modification time of path.

leo.core.leoGlobals.**os_path_getsize**(*path*)
Return the size of path.

leo.core.leoGlobals.**os_path_isabs**(*path*)
Return True if path is an absolute path.

leo.core.leoGlobals.**os_path_isdir**(*path*)
Return True if the path is a directory.

leo.core.leoGlobals.**os_path_isfile**(*path*)
Return True if path is a file.

leo.core.leoGlobals.**os_path_join**(**args*, ***keys*)
The same as os.path.join, but safe for unicode. In addition, it supports the !! and . conventions.

leo.core.leoGlobals.**os_path_normcase**(*path*)
Normalize the path's case.

leo.core.leoGlobals.**os_path_normpath**(*path*)
Normalize the path.

leo.core.leoGlobals.**os_path_normslashes**(*path*)

leo.core.leoGlobals.**os_path_realpath**(*path*)
Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

leo.core.leoGlobals.**os_path_split**(*path*)

leo.core.leoGlobals.**os_path_splitext**(*path*)

leo.core.leoGlobals.**os_startfile**(*fname*)

leo.core.leoGlobals.**pause**(*s*)

leo.core.leoGlobals.**pdb**(*message=''*)
Fall into pdb.

leo.core.leoGlobals.**pep8_class_name**(*s*)
Return the proper class name for s.

leo.core.leoGlobals.**pluginIsLoaded**(*fn*)

leo.core.leoGlobals.**plugin_date**(*plugin_mod*, *format=None*)

leo.core.leoGlobals.**plugin_signon**(*module_name*, *verbose=False*)

leo.core.leoGlobals.**plural**(*obj*)
Return "s" or "" depending on n.

leo.core.leoGlobals.**pno**(*tag=''*)
Print newly allocated objects.

leo.core.leoGlobals.**pr**(**args*, ***keys*)
Print all non-keyword args. This is a wrapper for the print statement.

The first, third, fifth, etc. arg translated by g.translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.core.leoGlobals.**prettyPrintType**(*obj*)

leo.core.leoGlobals.**printDict**(*obj*, *indent=''*, *printCaller=False*, *tag=None*)
Pretty print any Python object using g.pr.

leo.core.leoGlobals.**printDiffTime**(*message*, *start*)

leo.core.leoGlobals.**printEntireTree**(*c*, *tag=''*)

leo.core.leoGlobals.**printGc**(*tag=None*)

leo.core.leoGlobals.**printGcAll**(*tag=''*)

leo.core.leoGlobals.**printGcObjects**(*tag=''*)
Print newly allocated objects.

leo.core.leoGlobals.**printGcRefs**(*tag=''*)

leo.core.leoGlobals.**printGcSummary**(*tag=''*)

leo.core.leoGlobals.**printGcVerbose**(*tag=''*)

leo.core.leoGlobals.**printGlobals**(*message=None*)

leo.core.leoGlobals.**printLeoModules**(*message=None*)

leo.core.leoGlobals.**printList**(*obj*, *indent=''*, *printCaller=False*, *tag=None*)
Pretty print any Python object using g.pr.

leo.core.leoGlobals.**printNewObjects**(*tag=''*)
Print newly allocated objects.

leo.core.leoGlobals.**printObj**(*obj*, *indent=''*, *printCaller=False*, *tag=None*)
Pretty print any Python object using g.pr.

leo.core.leoGlobals.**printStack**()

leo.core.leoGlobals.**printStats**(*name=None*)

leo.core.leoGlobals.**printTuple**(*obj*, *indent=''*, *printCaller=False*, *tag=None*)
Pretty print any Python object using g.pr.

leo.core.leoGlobals.**print_bindings**(*name*, *window*)

leo.core.leoGlobals.**python_tokenize**(*s*, *line_numbers=True*)
Tokenize string s and return a list of tokens (kind,value,line_number)

where kind is in ('comment','id','nl','other','string','ws').

leo.core.leoGlobals.**rawPrint**(*s*)

leo.core.leoGlobals.**readFileIntoEncodedString**(*fn*, *silent=False*)
Return the raw contents of the file whose full path is fn.

leo.core.leoGlobals.**readFileIntoString**(*fn*, *encoding='utf-8'*, *kind=None*)
Return the contents of the file whose full path is fn.

Return (s,e) s is the string, converted to unicode, or None if there was an error. e is the encoding of s, computed in the following order: - The BOM encoding if the file starts with a BOM mark. - The encoding given in the # -- *coding: utf-8* -- line for python files. - The encoding given by the 'encoding' keyword arg. - None, which typically means 'utf-8'.

leo.core.leoGlobals.**readFileIntoUnicodeString**(*fn*, *encoding=None*, *silent=False*)
Return the raw contents of the file whose full path is fn.

leo.core.leoGlobals.**readlineForceUnixNewline**(*f*, *fileName=None*)

leo.core.leoGlobals.**recursiveUNLFind**(*unlList*, *c*, *depth=0*, *p=None*, *maxdepth=0*, *maxp=None*, *soft_idx=False*, *hard_idx=False*)
Internal part of recursiveUNLSearch which doesn't change the selected position or call c.frame.bringToFront()

returns found, depth, p, where:

- found is True if a full match was found
- depth is the depth of the best match
- p is the position of the best match

NOTE: maxdepth is max depth seen in recursion so far, not a limit on how far we will recurse. So it should default to 0 (zero).

- *unlList*: list of 'headline', 'headline:N', or 'headline:N,M' elements, where N is the node's position index and M the zero based count of like named nodes, eg. 'foo:2', 'foo:4,1', 'foo:12,3'
- *c*: outline
- *soft_idx*: use index when matching name not found
- *hard_idx*: use only indexes, ignore node names
- *depth*: part of recursion, don't set explicitly
- *p*: part of recursion, don't set explicitly
- *maxdepth*: part of recursion, don't set explicitly
- *maxp*: part of recursion, don't set explicitly

leo.core.leoGlobals.**recursiveUNLParts**(*text*)
recursiveUNLParts - return index, occurence, line_number, col_number from an UNL fragment. line_number is allowed to be negative to indicate a "global" line number within the file.

Parameters **text** (*str*) – the fragment, foo or foo:2 or foo:2,0,4,10

Returns index, occurence, line_number, col_number

Return type (int, int, int, int) or (None, None, None, None)

leo.core.leoGlobals.**recursiveUNLSearch**(*unlList*, *c*, *depth=0*, *p=None*, *maxdepth=0*, *maxp=None*, *soft_idx=False*, *hard_idx=False*)
try and move to unl in the commander c

All parameters passed on to recursiveUNLFind(), see that for docs.

NOTE: maxdepth is max depth seen in recursion so far, not a limit on how far we will recurse. So it should default to 0 (zero).

leo.core.leoGlobals.**red**(**args*, ***keys*)

leo.core.leoGlobals.**redirectStderr**()

leo.core.leoGlobals.**redirectStdout**()

leo.core.leoGlobals.**registerExclusiveHandler**(*tags*, *fn*)

leo.core.leoGlobals.**registerHandler**(*tags*, *fn*)

leo.core.leoGlobals.**regularizeTrailingNewlines**(*s*, *kind*)
Kind is 'asis', 'zero' or 'one'.

leo.core.leoGlobals.**removeBlankLines**(*s*)

leo.core.leoGlobals.**removeExtraLws**(*s*, *tab_width*)
Remove extra indentation from one or more lines.

Warning: used by getScript. This is *not* the same as g.adjustTripleString.

leo.core.leoGlobals.**removeLeading**(*s*, *chars*)
Remove all characters in chars from the front of s.

leo.core.leoGlobals.**removeLeadingBlankLines**(*s*)

leo.core.leoGlobals.**removeLeadingWhitespace**(*s*, *first_ws*, *tab_width*)

leo.core.leoGlobals.**removeTrailing**(*s*, *chars*)
Remove all characters in chars from the end of s.

leo.core.leoGlobals.**removeTrailingWs**(*s*)

leo.core.leoGlobals.**replace_path_expression**(*c*, *expr*)
local function to replace a single path expression.

leo.core.leoGlobals.**restoreStderr**()

leo.core.leoGlobals.**restoreStdout**()

leo.core.leoGlobals.**run_pylint**(*fn*, *rc*, *dots=True*, *patterns=None*, *sherlock=False*, *show_return=True*, *stats_patterns=None*, *verbose=True*)
Run pylint with the given args, with Sherlock tracing if requested.

Do not assume g.app exists.

run() in pylint-leo.py and PylintCommand.run_pylint *optionally* call this function.

leo.core.leoGlobals.**sanitize_filename**(*s*)
Prepares string s to be a valid file name:

- substitute '_' for whitespace and special path characters.
- eliminate all other non-alphabetic characters.
- convert double quotes to single quotes.
- strip leading and trailing whitespace.
- return at most 128 characters.

leo.core.leoGlobals.**scanAllAtPathDirectives**(*c*, *p*)

leo.core.leoGlobals.**scanAllAtTabWidthDirectives**(*c*, *p*)
Scan p and all ancestors looking for @tabwidth directives.

leo.core.leoGlobals.**scanAllAtWrapDirectives**(*c*, *p*)
Scan p and all ancestors looking for @wrap/@nowrap directives.

leo.core.leoGlobals.**scanAtCommentAndAtLanguageDirectives**(*aList*)
Scan aList for @comment and @language directives.

@comment should follow @language if both appear in the same node.

leo.core.leoGlobals.**scanAtEncodingDirectives**(*aList*)
Scan aList for @encoding directives.

leo.core.leoGlobals.**scanAtHeaderDirectives**(*aList*)
scan aList for @header and @noheader directives.

leo.core.leoGlobals.**scanAtLineendingDirectives**(*aList*)
Scan aList for @lineending directives.

leo.core.leoGlobals.**scanAtPagewidthDirectives**(*aList*, *issue_error_flag=False*)
Scan aList for @pagewidth directives.

leo.core.leoGlobals.**scanAtPathDirectives**(*c*, *aList*)

leo.core.leoGlobals.**scanAtRootDirectives**(*aList*)
Scan aList for @root directives.

leo.core.leoGlobals.**scanAtRootOptions**(*s*, *i*, *err_flag=False*)

leo.core.leoGlobals.**scanAtTabwidthDirectives**(*aList*, *issue_error_flag=False*)
Scan aList for @tabwidth directives.

leo.core.leoGlobals.**scanAtWrapDirectives**(*aList*, *issue_error_flag=False*)
Scan aList for @wrap and @nowrap directives.

leo.core.leoGlobals.**scanDirectives**(*c*, *p=None*)

leo.core.leoGlobals.**scanError**(*s*)
Bump the error count in the tangle command.

leo.core.leoGlobals.**scanForAtIgnore**(*c*, *p*)
Scan position p and its ancestors looking for @ignore directives.

leo.core.leoGlobals.**scanForAtLanguage**(*c*, *p*)
Scan position p and p's ancestors looking only for @language and @ignore directives.

Returns the language found, or c.target_language.

leo.core.leoGlobals.**scanForAtSettings**(*p*)
Scan position p and its ancestors looking for @settings nodes.

leo.core.leoGlobals.**scanf**(*s*, *pat*)

leo.core.leoGlobals.**setDefaultDirectory**(*c*, *p*, *importing=False*)
Return a default directory by scanning @path directives.

leo.core.leoGlobals.**setGlobalOpenDir**(*fileName*)

leo.core.leoGlobals.**set_delims_from_language**(*language*)
Return a tuple (single,start,end) of comment delims.

leo.core.leoGlobals.**set_delims_from_string**(*s*)
Return (delim1, delim2, delim2), the delims following the @comment directive.

This code can be called from @language logic, in which case s can point at @comment

leo.core.leoGlobals.**set_language**(*s*, *i*, *issue_errors_flag=False*)
Scan the @language directive that appears at s[i:].

The @language may have been stripped away.

Returns (language, delim1, delim2, delim3)

leo.core.leoGlobals.**shortFileName**(*fileName*, *n=None*)
Return the base name of a path.

leo.core.leoGlobals.**shortFilename**(*fileName*, *n=None*)
Return the base name of a path.

leo.core.leoGlobals.**skip_blank_lines**(*s*, *i*)

leo.core.leoGlobals.**skip_block_comment**(*s*, *i*)

leo.core.leoGlobals.**skip_braces**(*s*, *i*)
Skips from the opening to the matching brace.

If no matching is found i is set to len(s)

leo.core.leoGlobals.**skip_c_id**(*s*, *i*)

leo.core.leoGlobals.**skip_heredoc_string**(*s*, *i*)

leo.core.leoGlobals.**skip_id**(*s*, *i*, *chars=None*)

leo.core.leoGlobals.**skip_leading_ws**(*s*, *i*, *ws*, *tab_width*)

leo.core.leoGlobals.**skip_leading_ws_with_indent**(*s*, *i*, *tab_width*)
Skips leading whitespace and returns (i, indent),

- i points after the whitespace
- indent is the width of the whitespace, assuming tab_width wide tabs.

leo.core.leoGlobals.**skip_line**(*s*, *i*)

leo.core.leoGlobals.**skip_long**(*s*, *i*)
Scan s[i:] for a valid int. Return (i, val) or (i, None) if s[i] does not point at a number.

leo.core.leoGlobals.**skip_nl**(*s*, *i*)
Skips a single “logical” end-of-line character.

leo.core.leoGlobals.**skip_non_ws**(*s*, *i*)

leo.core.leoGlobals.**skip_parens**(*s*, *i*)
Skips from the opening (to the matching).

If no matching is found i is set to len(s)

leo.core.leoGlobals.**skip_pascal_begin_end**(*s*, *i*)
Skips from begin to matching end. If found, i points to the end. Otherwise, i >= len(s) The end keyword matches begin, case, class, record, and try.

leo.core.leoGlobals.**skip_pascal_block_comment**(*s*, *i*)

leo.core.leoGlobals.**skip_pascal_braces**(*s*, *i*)

leo.core.leoGlobals.**skip_pascal_string**(*s*, *i*)

leo.core.leoGlobals.**skip_pp_directive**(*s*, *i*)

leo.core.leoGlobals.**skip_pp_if**(*s*, *i*)

leo.core.leoGlobals.**skip_pp_part**(*s*, *i*)

leo.core.leoGlobals.**skip_python_string**(*s*, *i*, *verbose=True*)

leo.core.leoGlobals.**skip_string**(*s*, *i*, *verbose=True*)
Scan forward to the end of a string. New in Leo 4.4.2 final: give error only if verbose is True

leo.core.leoGlobals.**skip_to_char**(*s*, *i*, *ch*)

leo.core.leoGlobals.**skip_to_end_of_line**(*s*, *i*)

leo.core.leoGlobals.**skip_to_semicolon**(*s*, *i*)

leo.core.leoGlobals.**skip_to_start_of_line**(*s*, *i*)

leo.core.leoGlobals.**skip_typedef**(*s*, *i*)

leo.core.leoGlobals.**skip_ws**(*s*, *i*)

leo.core.leoGlobals.**skip_ws_and_nl**(*s*, *i*)

leo.core.leoGlobals.**sleep**(*n*)
Wait about n milliseconds.

leo.core.leoGlobals.**splitLines**(*s*)
Split s into lines, preserving the number of lines and the endings of all lines, including the last line.

leo.core.leoGlobals.**splitLongFileName**(*fn*, *limit=40*)
Return fn, split into lines at slash characters.

leo.core.leoGlobals.**splitlines**(*s*)
Split s into lines, preserving the number of lines and the endings of all lines, including the last line.

leo.core.leoGlobals.**startTracer**(*limit=0*, *trace=False*, *verbose=False*)

leo.core.leoGlobals.**stat**(*name=None*)
Increments the statistic for name in g.app.statsDict The caller's name is used by default.

leo.core.leoGlobals.**stdErrIsRedirected**()

leo.core.leoGlobals.**stdOutIsRedirected**()

leo.core.leoGlobals.**stripBOM**(*s*)
If there is a BOM, return (e,s2) where e is the encoding implied by the BOM and s2 is the s stripped of the BOM.

If there is no BOM, return (None,s)

s must be the contents of a file (a string) read in binary mode.

leo.core.leoGlobals.**stripBlankLines**(*s*)

leo.core.leoGlobals.**stripBrackets**(*s*)
Same as s.lstrip('<').rstrip('>') except it works for Python 2.2.1.

leo.core.leoGlobals.**stripPathCruft**(*path*)
Strip cruft from a path name.

leo.core.leoGlobals.**timeSince**(*start*)

leo.core.leoGlobals.**toEncodedString**(*s*, *encoding='utf-8'*, *reportErrors=False*)
Convert unicode string to an encoded string.

leo.core.leoGlobals.**toEncodedStringWithErrorCode**(*s*, *encoding*, *reportErrors=False*)
For unit testing: convert s to an encoded string and return (s,ok).

leo.core.leoGlobals.**toPythonIndex**(*s*, *index*)
Convert index to a Python int.

index may be a Tk index (x.y) or 'end'.

leo.core.leoGlobals.**toString**(*obj*, *indent=''*, *printCaller=False*, *tag=None*)
Pretty print any Python object to a string.

leo.core.leoGlobals.**toUnicode**(*s*, *encoding='utf-8'*, *reportErrors=False*)
Connvert a non-unicode string with the given encoding to unicode.

leo.core.leoGlobals.**toUnicodeFileEncoding**(*path*)

leo.core.leoGlobals.**toUnicodeWithErrorCode**(*s*, *encoding*, *reportErrors=False*)
For unit testing: convert s to unicode and return (s,ok).

leo.core.leoGlobals.**tr**(*s*)
Return the translated text of s.

leo.core.leoGlobals.**trace**(**args*, ***keys*)
Print a tracing message.

leo.core.leoGlobals.**traceUrl**(*c*, *path*, *parsed*, *url*)

leo.core.leoGlobals.**translateArgs**(*args*, *d*)
Return the concatenation of s and all args,

with odd args translated.

leo.core.leoGlobals.**translateString**(*s*)
Return the translated text of s.

leo.core.leoGlobals.**truncate**(*s*, *n*)
Return s truncated to n characters.

leo.core.leoGlobals.**tupleToString**(*obj*, *indent=''*, *tag=None*)
Pretty print a Python tuple to a string.

leo.core.leoGlobals.**u**(*s*)
Return s, converted to unicode from Qt widgets.

leo.core.leoGlobals.**ue**(*s*, *encoding*)

leo.core.leoGlobals.**unCamel**(*s*)
Return a list of sub-words in camelCased string s.

leo.core.leoGlobals.**unloadOnePlugin**(*moduleOrFileName*, *verbose=False*)

leo.core.leoGlobals.**unquoteUrl**(*url*)
Replace special characters (especially %20, by their equivalent).

This function handles 2/3 issues and suppresses pylint complaints.

leo.core.leoGlobals.**unregisterHandler**(*tags*, *fn*)

leo.core.leoGlobals.**update_file_if_changed**(*c*, *file_name*, *temp_name*)
Compares two files.

If they are different, we replace file_name with temp_name. Otherwise, we just delete temp_name. Both files should be closed.

leo.core.leoGlobals.**ustr**(*s*)
Define the pyzo ustr function.

leo.core.leoGlobals.**utils_chmod**(*fileName*, *mode*, *verbose=True*)

leo.core.leoGlobals.**utils_remove**(*fileName*, *verbose=True*)

leo.core.leoGlobals.**utils_rename**(*c*, *src*, *dst*, *verbose=True*)
Platform independent rename.

leo.core.leoGlobals.**utils_stat**(*fileName*)
Return the access mode of named file, removing any setuid, setgid, and sticky bits.

leo.core.leoGlobals.**virtual_event_name**(*s*)

leo.core.leoGlobals.**warning**(**args*, ***keys*)

leo.core.leoGlobals.**windows**()

leo.core.leoGlobals.**wrap_lines**(*lines*, *pageWidth*, *firstLineWidth=None*)
Returns a list of lines, consisting of the input lines wrapped to the given pageWidth.

1.2.1.21 leoGui Module

1.2.1.22 leoIPython Module

1.2.1.23 leoImport Module

class leo.core.leoImport.**FreeMindImporter**(*c*)
Bases: object

Importer class for FreeMind (.mmap) files.

add_children(*parent*, *element*)
parent is the parent position, element is the parent element. Recursively add all the child elements as descendants of parent_p.

create_outline(*path*)
Create a tree of nodes from a FreeMind file.

import_file(*path*)
The main line of the FreeMindImporter class.

import_files(*files*)
Import a list of FreeMind (.mmap) files.

prompt_for_files()
Prompt for a list of FreeMind (.mm.html) files and import them.

class leo.core.leoImport.**JSON_Import_Helper**(*c*)
Bases: object

A class that helps client scripts import .json files.

Client scripts supply data describing how to create Leo outlines from the .json data.

create_nodes(*parent*, *parent_d*)
Create the tree of nodes rooted in parent.

create_outline(*path*)

scan(*s*, *parent*)
Create an outline from a MindMap (.csv) file.

class leo.core.leoImport.**LeoImportCommands**(*c*)
Bases: object

A class implementing all of Leo's import/export code. This class uses **importers** in the leo/plugins/importers folder.

For more information, see leo/plugins/importers/howto.txt.

appendHeadRef(*p*, *file_name*, *head_ref*, *result*)

appendRefToFileName(*file_name*, *result*)

appendStringToBody(*p*, *s*)
Similar to c.appendStringToBody, but does not recolor the text or redraw the screen.

body_parser_for_ext(*ext*)
A factory returning a body parser function for the given file extension.

cSharpUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

cUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

coffeeScriptUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

compute_unit_test_kind(*ext*, *fn*)
Return kind from fn's file extension.

convertCodePartToWeb(*s*, *i*, *p*, *result*)
Headlines not containing a section reference are ignored in noweb and generate index index in cweb.

convertDocPartToWeb(*s*, *i*, *result*)

convertVnodeToWeb(*v*)
This code converts a VNode to noweb text as follows:

Convert @doc to @ Convert @root or @code to < < name > >=, assuming the headline contains < < name > > Ignore other directives Format doc parts so they fit in pagewidth columns. Output code parts as is.

copyPart(*s*, *i*, *result*)

createHeadline(*parent*, *body*, *headline*)
Create a new VNode as the last child of parent position.

createOutline(*fileName*, *parent*, *atShadow=False*, *ext=None*, *s=None*)
Create an outline by importing a file, reading the file with the given encoding if string s is None.

ext, The file extension to be used, or None. fileName: A string or None. The name of the file to be read. parent: The parent position of the created outline. s: A string or None. The file's contents.

createOutlineFromWeb(*path*, *parent*)

cstCanonicalize(*s*, *lower=True*)

cstDump()

cstEnter(*s*)

cstLookup(*target*)

ctextUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

dartUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

defaultImporterUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

dispatch(*ext*, *p*)
Return the correct scanner function for p, an @auto node.

elispUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

error(*s*)

exportHeadlines(*fileName*)

findFunctionDef(*s*, *i*)

flattenOutline(*fileName*)
A helper for the flatten-outline command.

Export the selected outline to an external file. The outline is represented in MORE format.

getFileName(*p*)
Return the file name from an @file or @root node.

getHeadRef(*p*)
Look for either noweb or cweb brackets. Return everything between those brackets.

get_import_filename(*fileName*, *parent*)
Return the absolute path of the file and set .default_directory.

htmlUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

importDerivedFiles(*parent=None*, *paths=None*, *command='Import'*)
Import one or more external files. This is not a command. It must *not* have an event arg. command is None when importing from the command line.

importFilesCommand(*files=None*, *parent=None*, *redrawFlag=True*, *shortFn=False*, *treeType=None*)

importFlattenedOutline(*files*)

importFreeMind(*files*)
Import a list of .mm.html files exported from FreeMind: http://freemind.sourceforge.net/wiki/index.php/Main_Page

importMindMap(*files*)
Import a list of .csv files exported from MindJet: https://www.mindjet.com/

importWebCommand(*files*, *webType*)

import_binary_file(*fileName*, *parent*)

iniUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

init_import(*atShadow*, *ext*, *fileName*, *s*)
Init ivars imports and read the file into s. Return ext, s.

isDocStart(*s*, *i*)

isModuleStart(*s*, *i*)

javaScriptUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

javaUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

languageForExtension(*ext*)
Return the language corresponding to the extension ext.

markdownUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

massageWebBody(*s*)

orgUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

otlUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

outlineToWeb(*fileName*, *webType*)

parse_body(*p*)
Parse p.b as source code, creating a tree of descendant nodes. This is essentially an import of p.b.

pascalUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

perlUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

phpUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

pythonUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

readAtAutoNodes()

reloadSettings()

reload_settings()

removeSentinelLines(*s*, *line_delim*, *start_delim*, *unused_end_delim*)
Properly remove all sentinle lines in s.

removeSentinelsCommand(*paths*, *toString=False*)

rstUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

scanBodyForHeadline(*s*)

scanUnknownFileType(*s*, *p*, *ext*)
Scan the text of an unknown file type.

scanWebFile(*fileName*, *parent*)

scannerUnitTest(*p*, *ext=None*, *fileName=None*, *s=None*, *showTree=False*)
Run a unit test of an import scanner, i.e., create a tree from string s at location p.

setBodyString(*p*, *s*)
Similar to c.setBodyString, but does not recolor the text or redraw the screen.

setEncoding(*p=None*, *default=None*)

textUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

typeScriptUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

weave(*filename*)

xmlUnitTest(*p*, *fileName=None*, *s=None*, *showTree=False*)

class leo.core.leoImport.**MORE_Importer**(*c*)
Bases: object

Class to import MORE files.

check(*s*)

check_lines(*strings*)

headlineLevel(*s*)
return the headline level of s,or -1 if the string is not a MORE headline.

import_file(*fileName*)

import_files(*files*)
Import a list of MORE (.csv) files.

import_lines(*strings*, *first_p*)

prompt_for_files()
Prompt for a list of MORE files and import them.

class leo.core.leoImport.**MindMapImporter**(*c*)
Bases: object

Mind Map Importer class.

create_outline(*path*)

csv_level(*row*)
Return the level of the given row.

csv_string(*row*)
Return the string for the given csv row.

import_files(*files*)
Import a list of MindMap (.csv) files.

prompt_for_files()
Prompt for a list of MindJet (.csv) files and import them.

scan(*path*, *target*)
Create an outline from a MindMap (.csv) file.

class leo.core.leoImport.**RecursiveImportController**(*c*, *kind*, *add_path=True*, *recursive=True*, *safe_at_file=True*, *theTypes=None*)
Bases: object

Recursively import all python files in a directory and clean the result.

add_class_names(*p*)
Add class names to headlines for all descendant nodes.

clear_dirty_bits(*p*)

dump_headlines(*p*)

fix_back_slashes(*p*)
Convert backslash to slash in all headlines.

import_dir(*dir_*, *parent*)
Import selected files from **dir_**, a directory.

import_one_file(*path*, *parent*)
Import one file to the last top-level node.

minimize_headlines(*p*, *prefix*)
Create @path nodes to minimize the paths required in descendant nodes.

post_process(*p*, *prefix*)
Traverse p's tree, replacing all nodes that start with prefix by the smallest equivalent @path or @file node.

remove_empty_nodes(*p*)
Remove empty nodes. Not called for @auto or @edit trees.

run(*dir_*)
Import all files whose extension matches self.theTypes in **dir_**. In fact, **dir_** can be a path to a single file.

strip_prefix(*path*, *prefix*)
Strip the prefix from the path and return the result.

class leo.core.leoImport.**TabImporter**(*c*, *separate=True*)
A class to import a file whose outline levels are indicated by leading tabs or blanks (but not both).

check(*lines*, *warn=True*)
Return False and warn if lines contains mixed leading tabs/blanks.

dump_stack()
Dump the stack, containing (level, p) tuples.

import_files(*files*)
Import a list of tab-delimited files.

lws(*s*)
Return the length of the leading whitespace of s.

prompt_for_files()
Prompt for a list of FreeMind (.mm.html) files and import them.

scan(*s1*, *fn=None*, *root=None*)
Create the outline corresponding to s1.

scan_helper(*s*)
Update the stack as necessary and return (level, parent, stack).

undent(*level*, *s*)
Unindent all lines of p.b by level.

class leo.core.leoImport.**ZimImportController**(*c*)
Bases: object

A class to import Zim folders and files: http://zim-wiki.org/ First use Zim to export your project to rst files.

Original script by Davy Cottet.

User options: @int rst_level = 0 @string rst_type @string zim_node_name @string path_to_zim

clean(*zimNode*, *rstType*)
Clean useless nodes

parseZimIndex()
Parse Zim wiki index.rst and return a list of tuples (level, name, path)

rstToLastChild(*p*, *name*, *rst*)
Import an rst file as a last child of pos node with the specified name

run()
Create the zim node as the last top-level node.

leo.core.leoImport.**headToPrevNode**(*event*)
Move the code preceding a def to end of previous node.

leo.core.leoImport.**import_MORE_files_command**(*event*)
Prompt for MORE files and import them.

leo.core.leoImport.**import_free_mind_files**(*event*)
Prompt for free-mind files and import them.

leo.core.leoImport.**import_mind_jet_files**(*event*)
Prompt for mind-jet files and import them.

leo.core.leoImport.**import_tabbed_files_command**(*event*)
Prompt for tabbed files and import them.

leo.core.leoImport.**import_zim_command**(*event*)
Import a zim folder, http://zim-wiki.org/, as the last top-level node of the outline.

First use Zim to export your project to rst files.

This command requires the following Leo settings:

```
@int rst_level = 0
@string rst_type
@string zim_node_name
@string path_to_zim
```

leo.core.leoImport.**parse_body_command**(*event*)
The parse-body command.

leo.core.leoImport.**tailToNextNode**(*event=None*)
Move the code following a def to start of next node.

1.2.1.24 leoInspect Module

1.2.1.25 leoKeys Module

Gui-independent keystroke handling for Leo.

class leo.core.leoKeys.**AutoCompleterClass**(*k*)
Bases: object

A class that inserts autocompleted and calltip text in text widgets. This class shows alternatives in the tabbed log pane.

The keyHandler class contains hooks to support these characters: invoke-autocompleter-character (default binding is '.') invoke-calltips-character (default binding is '(')

abort()

add_prefix(*prefix*, *s*)
A hack to match the callers expectations.

appendTabName(*word*)

attr_matches(*s*, *namespace*)
Compute matches when string s is of the form name.name....name.

Evaluates s using eval(s,namespace)

Assuming the text is of the form NAME.NAME....[NAME], and is evaluatable in the namespace, it will be evaluated and its attributes (as revealed by dir()) are used as possible completions.

For class instances, class members are are also considered.)

Warning: this can still invoke arbitrary C code, if an object with a __getattr__ hook is evaluated.

autoComplete(*event=None*, *force=False*)
An event handler for autocompletion.

autoCompleteForce(*event=None*)
Show autocompletion, even if autocompletion is not presently enabled.

auto_completer_state_handler(*event*)
Handle all keys while autocompleting.

beginTabName(*word*)

calltip()
Show the calltips for the present prefix. ch is '(' if the user has just typed it.

calltip_fail(*prefix*)
Evaluation of prefix failed.

calltip_success(*prefix*, *obj*)

clean(*hits*)
Clean up hits, a list of ctags patterns, for use in completion lists.

clean_completion_list(*header*, *tabList*)
Return aList with header removed from the start of each list item.

clean_for_display(*hits*)
Clean up hits, a list of ctags patterns, for display purposes.

clearTabName()

cmd()
Command decorator for the AutoCompleter class.

compute_completion_list()
Return the autocompleter completion list.

disableAutocompleter(*event=None*)
Disable the autocompleter.

disableCalltips(*event=None*)
Disable calltips.

do_backspace()
Delete the character and recompute the completion list.

do_qcompleter_tab(*prefix*, *options*)
Return the longest common prefix of all the options.

enableAutocompleter(*event=None*)
Enable the autocompleter.

enableCalltips(*event=None*)
Enable calltips.

exit()

finish()

get_autocompleter_prefix()

get_cached_options(*prefix*)

get_codewise_completions(*prefix*)
Use codewise to generate a list of hits.

get_completions(*prefix*)
Return jedi or codewise completions.

get_jedi_completions(*prefix*)

get_leo_completions(*prefix*)
Return completions in an environment defining c, g and p.

get_leo_namespace(*prefix*)
Return an environment in which to evaluate prefix. Add some common standard library modules as needed.

get_object()
Return the object corresponding to the current prefix.

get_summary_list(*header*, *tabList*)
Show the possible starting letters, but only if there are more than one.

guess_class(*c*, *varname*)
Return kind, class_list

info()
Show the docstring for the present completion.

init_qcompleter(*event=None*)

init_tabcompleter(*event=None*)

insert_general_char(*ch*)

insert_string(*s*, *select=False*)
Insert s at the insertion point.

is_leo_source_file()
Return True if this is one of Leo's source files.

jedi_warning = False

lookup_functions(*prefix*)

lookup_methods(*aList*, *prefix*)

lookup_modules(*aList*, *prefix*)

popTabName()

put(**args*, ***keys*)
Put s to the given tab.

May be overridden in subclasses.

reloadSettings()

setTabName(*s*)

showAutocompleterStatus()
Show the autocompleter status.

showCalltips(*event=None*, *force=False*)
Show the calltips at the cursor.

showCalltipsForce(*event=None*)
Show the calltips at the cursor, even if calltips are not presently enabled.

showCalltipsStatus()
Show the autocompleter status.

show_completion_list(*common_prefix*, *prefix*, *tabList*)

start(*event*)

strip_brackets(*s*)
Return s with all brackets removed.

This (mostly) ensures that eval will not execute function calls, etc.

toggleAutocompleter(*event=None*)
Toggle whether the autocompleter is enabled.

toggleCalltips(*event=None*)
Toggle whether calltips are enabled.

class leo.core.leoKeys.**ContextSniffer**
Bases: object

Class to analyze surrounding context and guess class

For simple dynamic code completion engines.

declare(*var*, *klass*)

get_classes(*s*, *varname*)
Return a list of classes for string s.

push_declarations(*s*)

class leo.core.leoKeys.**FileNameChooser**(*c*)
Bases: object

A class encapsulation file selection & completion logic.

compute_tab_list()
Compute the list of completions.

do_back_space()
Handle a back space.

do_char(*char*)
Handle a non-special character.

do_tab()
Handle tab completion.

extend_label(*s*)
Extend the label by s.

get_file_name(*event*, *callback*, *filterExt*, *prompt*, *tabName*)
Get a file name, supporting file completion.

get_label()
Return the label, not including the prompt.

set_label(*s*)
Set the label after the prompt to s. The prompt never changes.

show_tab_list(*tabList*)
Show the tab list in the log tab.

class leo.core.leoKeys.**GetArg**(*c*, *prompt='full-command: '*, *tabName='Completion'*)
Bases: object

A class encapsulating all k.getArg logic.

k.getArg maps to ga.get_arg, which gets arguments in the minibuffer.

For details, see the docstring for ga.get_arg

cancel_after_state()

command_source(*commandName*)
Return the source legend of an @button/@command node. 'G' leoSettings.leo 'M' myLeoSettings.leo 'L' local .leo File ' ' not an @command or @button node

compute_tab_list(*tabList*, *backspace=False*, *allow_empty_completion=False*)
Compute and show the available completions.

do_back_space(*tabList*, *completion=True*)
Handle a backspace and update the completion list.

do_char(*event*, *char*)
Handle a non-special character.

do_end(*event*, *char*, *stroke*)
A return or escape has been seen.

do_state_zero(*completion*, *event*, *handler*, *oneCharacter*, *returnKind*, *returnState*, *tabList*, *useMinibuffer*)
Do state 0 processing.

do_tab(*tabList*, *completion=True*)
Handle tab completion when the user hits a tab.

do_tab_callback()
If the command-name handler has a tab_callback, call handler.tab_callback() and return True.

do_tab_cycling(*common_prefix*, *tabList*)
Put the next (or first) completion in the minibuffer.

get_arg(*event*, *returnKind=None*, *returnState=None*, *handler=None*, *tabList=None*, *completion=True*, *oneCharacter=False*, *stroke=None*, *useMinibuffer=True*)
Accumulate an argument. Enter the given return state when done.

Ctrl-G will abort this processing at any time.

All commands needing user input call k.getArg, which just calls ga.get_arg.

The arguments to ga.get_arg are as follows:

event: The event passed to the command.

returnKind=None: A string. returnState=None, An int. handler=None, A function.

When the argument is complete, ga.do_end does:

```
if kind: k.setState(kind,n,handler)
```

tabList=[]: A list of possible completions.

completion=True: True if completions are enabled.

oneCharacter=False: True if k.arg should be a single character.

stroke=None: The incoming key stroke.

useMinibuffer=True: True: put focus in the minibuffer while accumulating arguments. False allows sort-lines, for example, to show the selection range.

get_command(*s*)
Return the command part of a minibuffer contents s.

get_label()
Return the label, not including the prompt.

get_minibuffer_command_name()
Return the command name in the minibuffer.

is_command(*s*)
Return False if something, even a blank, follows a command.

reset_tab_cycling()
Reset all tab cycling ivars.

set_label(*s*)
Set the label after the prompt to s. The prompt never changes.

should_end(*char*, *stroke*)
Return True if ga.get_arg should return.

show_tab_list(*tabList*)
Show the tab list in the log tab.

trace_state(*char*, *completion*, *handler*, *state*, *stroke*)
Trace the vars and ivars.

class leo.core.leoKeys.**KeyHandlerClass**(*c*)
Bases: object

A class to support emacs-style commands. c.k is an instance of this class.

NEWgeneralModeHandler(*event*, *commandName=None*, *func=None*, *modeName=None*, *nextMode=None*, *prompt=None*)
Handle a mode defined by an @mode node in leoSettings.leo.

addModeCommands()
Add commands created by @mode settings to c.commandsDict.

addToCommandHistory(*commandName*)
Add a name to the command history.

badMode(*modeName*)

bindKey(*pane*, *shortcut*, *callback*, *commandName*, *modeFlag=False*, *tag=None*)
Bind the indicated shortcut (a Tk keystroke) to the callback.

No actual gui bindings are made: only entries in k.masterBindingsDict and k.bindingsDict.

tag gives the source of the binding.

Return True if the binding was made successfully.

bindKeyToDict(*pane*, *stroke*, *bi*)
Update k.masterBindingsDict for the stroke.

bindOpenWith(*d*)
Register an open-with command.

bindShortcut(*pane*, *shortcut*, *callback*, *commandName*, *modeFlag=False*, *tag=None*)
Bind the indicated shortcut (a Tk keystroke) to the callback.

No actual gui bindings are made: only entries in k.masterBindingsDict and k.bindingsDict.

tag gives the source of the binding.

Return True if the binding was made successfully.

callAltXFunction(*event*)
Call the function whose name is in the minibuffer.

callStateFunction(*event*)
Call the state handler associated with this event.

checkBindings()
Print warnings if commands do not have any @shortcut entry. The entry may be *None*, of course.

checkKeyEvent(*event*)
Perform sanity checks on the incoming event.

check_bind_key(*commandName*, *pane*, *stroke*)
Return True if the binding of stroke to commandName for the given pane can be made.

clearState()
Clear the key handler state.

cmd()
Command decorator for the leoKeys class.

commandExists(*commandName*)
Return the command handler for the given command name, or None.

commandHistoryBackwd()
Return the previous entry in the Command History - stay at the top if we are there

commandHistoryFwd()
Move down the Command History - fall off the bottom (return empty string) if necessary

completeAllBindings(*w=None*)
New in 4.4b3: make an actual binding in *all* the standard places.

The event will go to k.masterKeyHandler as always, so nothing really changes. except that k.masterKeyHandler will know the proper stroke.

completeAllBindingsForWidget(*w*)
Make all a master gui binding for widget w.

computeInverseBindingDict()

createModeBindings(*modeName*, *d*, *w*)
Create mode bindings for the named mode using dictionary d for w, a text widget.

defineExternallyVisibleIvars()

defineInternalIvars()
Define internal ivars of the KeyHandlerClass class.

defineMultiLineCommands()

defineSingleLineCommands()

doBackSpace(*tabList*, *completion=True*)
Convenience method mapping k.doBackSpace to ga.do_back_space.

doBinding(*event*)
The last phase of k.masertKeyHandler. Execute the command associated with stroke's binding. Call k.handleUnboundKeys for killed or non-existent bindings.

doControlU(*event*, *stroke*)

doDemo(*event*)
Support the demo.py plugin. Return True if k.masterKeyHandler should return.

doKeyboardQuit(*event*)
Handle keyboard-quit logic. return True if k.masterKeyHandler should return.

doMode(*event*)
Handle mode bindings. Return True if k.masterKeyHandler should return.

doTabCompletion(*tabList*)
Convenience method mapping k.doTabCompletion to ga.do_tab.

doUnboundPlainKey(*event*)
Handle unbound plain keys. Return True if k.masterKeyHandler should return.

doVim(*event*)
Handle vim mode. Return True if k.masterKeyHandler should return.

dumpMasterBindingsDict()
Dump k.masterBindingsDict.

editShortcut_do_bind_helper(*stroke*, *cmdname*)

endCommand(*commandName*)
Make sure Leo updates the widget following a command.

Never changes the minibuffer label: individual commands must do that.

endMode()

enterNamedMode(*event*, *commandName*)

executeNTimes(*event*, *n*)

exitNamedMode(*event=None*)
Exit an input mode.

extendLabel(*s*, *select=False*, *protect=False*)

finishCreate()
Complete the construction of the keyHandler class. c.commandsDict has been created when this is called.

fullCommand(*event*, *specialStroke=None*, *specialFunc=None*, *help=False*, *helpHandler=None*)
Handle 'full-command' (alt-x) mode.

generalModeHandler(*event*, *commandName=None*, *func=None*, *modeName=None*, *nextMode=None*, *prompt=None*)
Handle a mode defined by an @mode node in leoSettings.leo.

get1Arg(*event*, *handler*, *prefix=None*, *tabList=None*, *completion=True*, *oneCharacter=False*, *stroke=None*, *useMinibuffer=True*)
k.get1Arg: Handle the next character the user types when accumulating a user argument from the minibuffer. Ctrl-G will abort this processing at any time.

Commands should use k.get1Arg to get the first minibuffer argument and k.getNextArg to get all other arguments.

Before going into the many details, let's look at some examples. This code will work in any class having a 'c' ivar bound to a commander.

Example 1: get one argument from the user:

@g.command('my-command') def myCommand(self, event):

k = self.c.k k.setLabelBlue('prompt: ') k.get1Arg(event, handler=self.myCommand1)

def myCommand1(self, event): k = self.c.k # k.arg contains the argument. # Finish the command. . . . # Reset the minibuffer. k.clearState() k.resetLabel() k.showStateAndMode()

Example 2: get two arguments from the user:

@g.command('my-command') def myCommand(self, event):

k = self.c.k k.setLabelBlue('first prompt: ') k.get1Arg(event, handler=self.myCommand1)

def myCommand1(self, event): k = self.c.k self.arg1 = k.arg k.extendLabel(' second prompt: ', select=False, protect=True) k.getNextArg(handler=self.myCommand2)

def myCommand2(self, event): k = self.c.k # k.arg contains second argument. # Finish the command, using self.arg1 and k.arg. . . . # Reset the minibuffer. k.clearState() k.resetLabel() k.showStateAndMode()

k.get1Arg and k.getNextArg are a convenience methods. They simply pass their arguments to the get_arg method of the singleton GetArg instance. This docstring describes k.get1arg and k.getNextArg as if they were the corresponding methods of the GetArg class.

k.get1Arg is a state machine. Logically, states are tuples (kind, n, handler) though they aren't represented that way. When the state machine in the GetArg class is active, the kind is 'getArg'. This constant has special meaning to Leo's key-handling code.

The arguments to k.get1Arg are as follows:

event: The event passed to the command.

handler=None, An executable. k.get1arg calls handler(event) when the user completes the argument by typing <Return> or (sometimes) <tab>.

tabList=[]: A list of possible completions.

completion=True: True if completions are enabled.

oneCharacter=False: True if k.arg should be a single character.

stroke=None: The incoming key stroke.

useMinibuffer=True: True: put focus in the minibuffer while accumulating arguments. False allows sort-lines, for example, to show the selection range.

getArg(*event*, *returnKind=None*, *returnState=None*, *handler=None*, *prefix=None*, *tabList=None*, *completion=True*, *oneCharacter=False*, *stroke=None*, *useMinibuffer=True*)
Convenience method mapping k.getArg to ga.get_arg.

getBindingHelper(*key*, *name*, *stroke*, *w*)
Find a binding for the widget with the given name.

getEditableTextRange()

getFileName(*event*, *callback=None*, *filterExt=None*, *prompt='Enter File Name: '*, *tabName='Dired'*)
Get a file name from the minibuffer.

getLabel(*ignorePrompt=False*)

getMinibufferCommandName()
Convenience method mapping k.getMinibufferCommandName to ga.get_minibuffer_command_name.

getNextArg(*handler*)
Get the next arg. For example, after a Tab in the find commands. See the docstring for k.get1Arg for examples of its use.

getPaneBinding(*stroke*, *w*)

getState(*kind*)

getStateHandler()

getStateKind()

getStrokeForCommandName(*commandName*)

handleDefaultChar(*event*, *stroke*)
Handle an unbound key, based on the event's widget. Do not assume that stroke exists.

handleInputShortcut(*event*, *stroke*)

handleMiniBindings(*event*, *state*, *stroke*)
Find and execute commands bound to the event.

handleMinibufferHelper(*event*, *pane*, *state*, *stroke*)
Execute a pane binding in the minibuffer.

Return 'continue', 'ignore', 'found'

handleUnboundKeys(*event*)

inState(*kind=None*)

initAbbrev()

initCommandHistory()
Init command history from @data command-history nodes.

initMode(*event*, *modeName*)

initOneAbbrev(*commandName*, *key*)
Enter key as an abbreviation for commandName in c.commandsDict.

initSpecialIvars()
Set ivars for special keystrokes from previously-existing bindings.

isAutoCompleteChar(*stroke*)
Return True if stroke is bound to the auto-complete in the insert or overwrite state.

isEditShortcutSensible()

isFKey(*stroke*)

isInShortcutBodyLine()

isPlainKey(*stroke*)
Return true if the shortcut refers to a plain (non-Alt,non-Ctl) key.

isSpecialKey(*event*)
Return True if char is a special key.

keyboardQuit(*event=None*, *setFocus=True*, *mouseClick=False*)
This method clears the state and the minibuffer label.

k.endCommand handles all other end-of-command chores.

killLine(*protect=True*)

kill_one_shortcut(*stroke*)
Update the *configuration* dicts so that c.config.getShortcut(name) will return None for all names *presently* bound to the stroke.

makeAllBindings()
Make all key bindings in all of Leo's panes.

makeBindingsFromCommandsDict()
Add bindings for all entries in c.commandsDict.

makeMasterGuiBinding(*stroke*, *w=None*, *trace=False*)
Make a master gui binding for stroke in pane w, or in all the standard widgets.

manufactureKeyPressForCommandName(*w*, *commandName*)
Implement a command by passing a keypress to the gui.

Only unit tests use this method.

masterCommand(*commandName=None*, *event=None*, *func=None*, *stroke=None*)
This is the central dispatching method. All commands and keystrokes pass through here. This returns None, but may set k.funcReturn.

masterKeyHandler(*event*)
The master key handler for almost all key bindings.

menuCommandKey(*event=None*)

modeHelp(*event*)
The mode-help command.

A possible convention would be to bind <Tab> to this command in most modes, by analogy with tab completion.

modeHelpHelper(*d*)

oops()

overrideCommand(*commandName*, *func*)

prettyPrintKey(*stroke*, *brief=False*)

printBindings(*event=None*)
Print all the bindings presently in effect.

printBindingsHelper(*result*, *data*, *prefix*)

printButtons(*event=None*)
Print all @button and @command commands, their bindings and their source.

printCommands(*event=None*)
Print all the known commands and their bindings, if any.

protectLabel()

registerCommand(*commandName*, *func*, *allowBinding=False*, *pane='all'*, *shortcut=None*, ***kwargs*)
Make the function available as a minibuffer command.

You can wrap any method in a callback function, so the restriction to functions is not significant.

Ignore the 'shortcut' arg unless 'allowBinding' is True.

Only k.bindOpenWith and the mod_scripting.py plugin should set allowBinding.

registerCommandShortcut(*commandName*, *func*, *pane*, *shortcut*)
Register a shortcut for the a command.

Important: Bindings created here from plugins can not be overridden. This includes @command and @button bindings created by mod_scripting.py.

reinitMode(*modeName*)

reloadSettings()

remove_conflicting_definitions(*aList*, *commandName*, *pane*, *shortcut*)

repeatComplexCommand(*event*)
Repeat the previously executed minibuffer command.

repeatComplexCommandHelper(*event*)

resetCommandHistory()
reset the command history index to indicate that we are pointing 'past' the last entry

resetLabel()
Reset the minibuffer label.

searchTree(*char*)
Search all visible nodes for a headline starting with stroke.

selectAll()
Select all the user-editable text of the minibuffer.

setCommandState(*event*)
Enter the 'command' editing state.

setDefaultEditingAction()

setDefaultInputState()

setDefaultUnboundKeyAction(*allowCommandState=True*)

setEditingState()

setEventWidget(*event*)
A hack: redirect the event to the text part of the log.

setInputState(*state*, *set_border=False*)

setInsertState(*event*)
Enter the 'insert' editing state.

setLabel(*s*, *protect=False*)
Set the label of the minibuffer.

setLabelBlue(*label*, *protect=True*)
Set the minibuffer label.

setLabelGray(*label=None*)

setLabelGrey(*label=None*)

setLabelRed(*label=None*, *protect=False*)

setLossage(*ch*, *stroke*)

setOverwriteState(*event*)
Enter the 'overwrite' editing state.

setState(*kind*, *n*, *handler=None*)

setStatusLabel(*s*)
Set the label to s.

Use k.setStatusLabel, not k.setLael, to report the status of a Leo command. This allows the option to use g.es instead of the minibuffer to report status.

showStateAndMode(*w=None*, *prompt=None*, *setFocus=True*)
Show the state and mode at the start of the minibuffer.

showStateCursor(*state*, *w*)

simulateCommand(*commandName*, *event=None*)
Execute a Leo command by name.

sortCommandHistory()
Sort the command history.

stroke2char(*stroke*)
Convert a stroke to an (insertable) char. This method allows Leo to use strokes everywhere.

toggleInputState(*event=None*)
The toggle-input-state command.

traceVars(*event*)

universalDispatcher(*event*)
Handle accumulation of universal argument.

updateLabel(*event*)
Mimic what would happen with the keyboard and a Text editor instead of plain accumulation.

class leo.core.leoKeys.**ModeInfo**(*c*, *name*, *aList*)
Bases: object

computeModeName(*name*)

computeModePrompt(*name*)

createModeBindings(*w*)
Create mode bindings for w, a text widget.

createModeCommand()

enterMode()

init(*name*, *dataList*)
aList is a list of tuples (commandName,bi).

initMode()

1.2.1.26 leoMenu Module

Gui-independent menu handling for Leo.

class leo.core.leoMenu.**LeoMenu**(*frame*)
Bases: object

The base class for all Leo menus.

activateMenu(*menuName*)

add_cascade(*parent*, *label*, *menu*, *underline*)

add_command(*menu*, ***keys*)

add_separator(*menu*)

canonicalizeMenuName(*name*)

canonicalizeTranslatedMenuName(*name*)

capitalizeMinibufferMenuName(*s*, *removeHyphens*)

clearAccel(*menu*, *name*)

computeOldStyleShortcutKey(*s*)
Compute the old-style shortcut key for @shortcuts entries.

createMasterMenuCallback(*dynamicMenu*, *command*, *commandName*)

createMenuBar(*frame*)

createMenuEntries(*menu*, *table*, *dynamicMenu=False*)
Create a menu entry from the table. New in 4.4: this method shows the shortcut in the menu, but this method **never** binds any shortcuts.

createMenuFromConfigList(*parentName*, *aList*, *level=0*)
Build menu based on nested list

List entries are either:

['@item', 'command-name', 'optional-view-name']

or:

['@menu Submenu name', <nested list>, None]

Parameters

- **parentName** (*str*) – name of menu under which to place this one
- **aList** (*list*) – list of entries as described above

createMenuItemsFromTable(*menuName*, *table*, *dynamicMenu=False*)

createMenusFromConfigList(*aList*)
Create menus from aList. The 'top' menu has already been created.

createMenusFromTables()
(leoMenu) Usually over-ridden.

createNewMenu(*menuName*, *parentName='top'*, *before=None*)

createOpenWithMenu(*parent*, *label*, *index*, *amp_index*)

createOpenWithMenuFromTable(*table*)
Table is a list of dictionaries, created from @openwith settings nodes.

This menu code uses these keys:

'name': menu label. 'shortcut': optional menu shortcut.

efc.open_temp_file uses these keys:

'args': the command-line arguments to be used to open the file. 'ext': the file extension. 'kind': the method used to open the file, such as subprocess.Popen.

createOpenWithMenuItemsFromTable(*menu*, *table*)
Create an entry in the Open with Menu from the table, a list of dictionaries.

Each dictionary d has the following keys:

'args': the command-line arguments used to open the file. 'ext': not used here: used by efc.open_temp_file. 'kind': not used here: used by efc.open_temp_file. 'name': menu label. 'shortcut': optional menu shortcut.

defineMenuCallback(*command*, *name*, *minibufferCommand*)

defineOpenWithMenuCallback(*d*)

define_enable_dict()

delete(*menu*, *realItemName*)

deleteMenu(*menuName*)

deleteMenuItem(*itemName*, *menuName='top'*)
Delete itemName from the menu whose name is menuName.

deleteRecentFilesMenuItems(*menu*)
Delete recent file menu entries

delete_range(*menu*, *n1*, *n2*)

destroy(*menu*)

destroyMenu(*menuName*)

disableMenu(*menu*, *name*)

enableMenu(*menu*, *name*, *val*)

error(*s*)

finishCreate()

getMacHelpMenu(*table*)

getMenu(*menuName*)

getMenuEntryBindings(*command*, *dynamicMenu*, *label*)
Compute commandName from command.

getMenuEntryInfo(*data*, *menu*)

getMenuLabel(*menu*, *name*)

getRealMenuName(*menuName*)

handleSpecialMenus(*name*, *parentName*, *alt_name=None*, *table=None*)
Handle a special menu if name is the name of a special menu. return True if this method handles the menu.

hasSelection()

insert(*menuName*, *position*, *label*, *command*, *underline=None*)

insert_cascade(*parent*, *index*, *label*, *menu*, *underline*)

new_menu(*parent*, *tearoff=0*, *label=''*)

oops()

setMenu(*menuName*, *menu*)

setMenuLabel(*menu*, *name*, *label*, *underline=-1*)

setRealMenuName(*untrans*, *trans*)

setRealMenuNamesFromTable(*table*)

traceMenuTable(*table*)

class leo.core.leoMenu.**NullMenu**(*frame*)
Bases: *leo.core.leoMenu.LeoMenu*

A null menu class for testing and batch execution.

oops()

1.2.1.27 leoNodes Module

Leo's fundamental data classes.

class leo.core.leoNodes.**NodeIndices**(*id_*)
Bases: object

A class managing global node indices (gnx's).

check_gnx(*c*, *gnx*, *v*)
Check that no vnode exists with the given gnx in fc.gnxDict.

compute_last_index(*c*)
Scan the entire leo outline to compute ni.last_index.

getDefaultId()
Return the id to be used by default in all gnx's

getNewIndex(*v*, *cached=False*)
Create a new gnx for v or an empty string if the hold flag is set. **Important**: the method must allocate a new gnx even if v.fileIndex exists.

new_vnode_helper(*c*, *gnx*, *v*)
Handle all gnx-related tasks for VNode.__init__.

scanGnx(*s*, *i=0*)
Create a gnx from its string representation.

setDefaultId(*theId*)
Set the id to be used by default in all gnx's

setTimeStamp()
Set the timestamp string to be used by getNewIndex until further notice

setTimestamp()
Set the timestamp string to be used by getNewIndex until further notice

tupleToString(*aTuple*)
Convert a gnx tuple returned by scanGnx to its string representation.

update()
Update self.timeString and self.lastIndex

updateLastIndex(*gnx*)
Update ni.lastIndex if the gnx affects it.

class leo.core.leoNodes.**PosList**
Bases: list

children()
Return a PosList instance containing pointers to all the immediate children of nodes in PosList self.

filter_b(*regex*, *flags=2*)
Find all the nodes in PosList self where body matches regex one or more times.

filter_h(*regex*, *flags=2*)
Find all the nodes in PosList self where zero or more characters at the beginning of the headline match regex

class leo.core.leoNodes.**Position**(*v*, *childIndex=0*, *stack=None*)
Bases: object

anyAtFileNodeName()

archivedPosition(*root_p=None*)
Return a representation of a position suitable for use in .leo files.

atAsisFileNodeName()

atAutoNodeName()

atCleanNodeName()

atEditNodeName()

atFileNodeName()

atNoSentFileNodeName()

atNoSentinelsFileNodeName()

atShadowFileNodeName()

atSilentFileNodeName()

atThinFileNodeName()

b
position body string property

back()

badUnlink(*parent_v*, *n*, *child*)

bodyString()

checkVisBackLimit(*limit*, *limitIsVisible*, *p*)
Return done, p or None

checkVisNextLimit(*limit*, *p*)
Return True is p is outside limit of visible nodes.

childIndex()

children()
Yield all child positions of p.

children_iter()
Yield all child positions of p.

cleanHeadString()

clearAllVisitedInTree()

clearDirty()

16. Set p.v dirty.

clearMarked()

clearOrphan()

clearVisited()

clearVisitedInTree()

clone()
Create a clone of back.

Returns the newly created position.

computeIcon()

contract()
Contract p.v and clear p.v.expandedPositions list.

convertTreeToString()
Convert a positions suboutline to a string in MORE format.

copy()
“Return an independent copy of a position.

copyTreeAfter(*copyGnxs=False*)
Copy p and insert it after itself.

copyTreeFromSelfTo(*p2*, *copyGnxs=False*)

copyWithNewVnodes(*copyMarked=False*)
Return an **unliked** copy of p with a new vnode v. The new vnode is complete copy of v and all its descendants.

createNodeHierarchy(*heads*, *forcecreate=False*)
Create the proper hierarchy of nodes with headlines defined in 'heads' as children of the current position

params: heads - list of headlines in order to create, i.e. ['foo','bar','baz']

will create: self -foo –bar —baz

forcecreate - If False (default), will not create nodes unless they don't exist If True, will create nodes regardless of existing nodes

returns the final position ('baz' in the above example)

deleteAllChildren()
Delete all children of the receiver.

directParents()

doDelete(*newNode=None*)
Deletes position p from the outline.

dump(*label=''*)

dumpLink(*link*)

expand()

findAllPotentiallyDirtyNodes()

findRootPosition()

firstChild()

following_siblings()
Yield all siblings positions that follow p, not including p.

following_siblings_iter()
Yield all siblings positions that follow p, not including p.

getBack()

getFirstChild()

getLastChild()

getLastNode()

getNext()

getNodeAfterTree()

getNthChild(*n*)

getParent()

getThreadBack()

getThreadNext()

getVisBack(*c*)

getVisNext(*c*)

get_UNL(*with_file=True*, *with_proto=False*, *with_index=True*, *with_count=False*)
with_file=True - include path to Leo file with_proto=False - include 'file://' with_index - include ',x' at end where x is child index in parent with_count - include ',x,y' at end where y zero based count of same headlines

gnx
position gnx property

h
position property returning the headline string

hasBack()

hasChildren()

hasFirstChild()

hasNext()

hasParent()

hasThreadBack()

hasThreadNext()

hasVisBack(*c*)

hasVisNext(*c*)

headString()

inAtIgnoreRange()
Returns True if position p or one of p's parents is an @ignore node.

in_at_all_tree()
Return True if p or one of p's ancestors is an @all node.

in_at_ignore_tree()
Return True if p or one of p's ancestors is an @ignore node.

initBodyString(*s*)

initExpandedBit()

initHeadString(*s*)

initMarkedBit()

initStatus(*status*)

insertAfter()
Inserts a new position after self.

Returns the newly created position.

insertAsLastChild()
Inserts a new VNode as the last child of self.

Returns the newly created position.

insertAsNthChild(*n*)
Inserts a new node as the the nth child of self. self must have at least n-1 children.

Returns the newly created position.

insertBefore()
Inserts a new position before self.

Returns the newly created position.

invalidOutline(*message*)

isAncestorOf(*p2*)
Return True if p is one of the direct ancestors of p2.

isAnyAtFileNode()

isAtAllNode()

isAtAsisFileNode()

isAtAutoNode()

isAtAutoRstNode()

isAtCleanNode()

isAtEditNode()

isAtFileNode()

isAtIgnoreNode()

isAtNoSentFileNode()

isAtNoSentinelsFileNode()

isAtOthersNode()

isAtRstFileNode()

isAtShadowFileNode()

isAtSilentFileNode()

isAtThinFileNode()

isCloned()

isDirty()

isExpanded()

isMarked()

isOrphan()

isOutsideAnyAtFileTree()
Select the first clone of target that is outside any @file node.

isRoot()

isSelected()

isTopBitSet()

isVisible(*c*)
Return True if p is visible in c's outline.

isVisited()

is_at_all()
Return True if p.b contains an @all directive.

is_at_ignore()
Return True if p is an @ignore node.

key()

lastChild()

lastNode()

level()
Return the number of p's parents.

matchHeadline(*pattern*)

moreBody()
Returns the body string in MORE format.

Inserts a backslash before any leading plus, minus or backslash.

moreHead(*firstLevel*, *useVerticalBar=False*)
Return the headline string in MORE format.

moveAfter(*a*)
Move a position after position a.

moveToBack()
Move self to its previous sibling.

moveToFirstChild()
Move a position to it's first child's position.

moveToFirstChildOf(*parent*)
Move a position to the first child of parent.

moveToLastChild()
Move a position to it's last child's position.

moveToLastChildOf(*parent*)
Move a position to the last child of parent.

moveToLastNode()
Move a position to last node of its tree.

N.B. Returns p if p has no children.

moveToNext()
Move a position to its next sibling.

moveToNodeAfterTree()
Move a position to the node after the position's tree.

moveToNthChild(*n*)

moveToNthChildOf(*parent*, *n*)
Move a position to the nth child of parent.

moveToParent()
Move a position to its parent position.

moveToRoot(*oldRoot=None*)
Moves a position to the root position.

Important: oldRoot must the previous root position if it exists.

moveToThreadBack()
Move a position to it's threadBack position.

moveToThreadNext()
Move a position to threadNext position.

moveToVisBack(*c*)
Move a position to the position of the previous visible node.

moveToVisNext(*c*)
Move a position to the position of the next visible node.

nearest(*predicate=None*)
A generator yielding all unique root positions "near" p1 = self that satisfy the given predicate. p.isAnyAtFileNode is the default predicate.

The search first proceeds up the p's tree. If a root is found, this generator yields just that root.

Otherwise, the generator yields all unique nodes in p.subtree() that satisfy the predicate. Once a root is found, the generator skips its subtree.

nearest_roots(*predicate=None*)
A generator yielding all the root positions "near" p1 = self that satisfy the given predicate. p.isAnyAtFileNode is the default predicate.

The search first proceeds up the p's tree. If a root is found, this generator yields just that root.

Otherwise, the generator yields all nodes in p.subtree() that satisfy the predicate. Once a root is found, the generator skips its subtree.

nearest_unique_roots(*predicate=None*)
A generator yielding all unique root positions "near" p1 = self that satisfy the given predicate. p.isAnyAtFileNode is the default predicate.

The search first proceeds up the p's tree. If a root is found, this generator yields just that root.

Otherwise, the generator yields all unique nodes in p.subtree() that satisfy the predicate. Once a root is found, the generator skips its subtree.

next()

nodeAfterTree()

nodes()
Yield p.v and all vnodes in p's subtree.

nosentinels
position property returning the body text without sentinels

nthChild(*n*)

numberOfChildren()

parent()

parents()
Yield all parent positions of p.

parents_iter()
Yield all parent positions of p.

positionAfterDeletedTree()
Return the position corresponding to p.nodeAfterTree() after this node is deleted. This will be p.nodeAfterTree() unless p.next() exists.

This method allows scripts to traverse an outline, deleting nodes during the traversal. The pattern is:

```
p = c.rootPosition()
while p:
if <delete p?>:
    next = p.positionAfterDeletedTree()
    p.doDelete()
    p = next
else:
    p.moveToThreadNext()
```

This method also allows scripts to *move* nodes during a traversal, **provided** that nodes are moved to a "safe" spot so that moving a node does not change the position of any other nodes.

For example, the move-marked-nodes command first creates a **move node**, called 'Clones of marked nodes'. All moved nodes become children of this move node. **Inserting** these nodes as children of the "move node" does not change the positions of other nodes. **Deleting** these nodes *may* change the position of nodes, but the pattern above handles this complication cleanly.

promote()
A low-level promote helper.

restoreCursorAndScroll()

safeMoveToThreadNext()
Move a position to threadNext position. Issue an error if any vnode is an ancestor of itself.

saveCursorAndScroll()

script
position property returning the script formed by p and its descendants

scriptSetBodyString(*s*)

self_and_parents()
Yield p and all parent positions of p.

self_and_parents_iter()
Yield p and all parent positions of p.

self_and_siblings()
Yield all sibling positions of p including p.

self_and_siblings_iter()
Yield all sibling positions of p including p.

self_and_subtree()
Yield p and all positions in p's subtree.

self_and_subtree_iter()
Yield p and all positions in p's subtree.

setAllAncestorAtFileNodesDirty(*setDescendentsDirty=False*)

setBodyString(*s*)

setDirty(*setDescendentsDirty=True*)
Mark a node and all ancestor @file nodes dirty.

Warning: p.setDirty() is *expensive* because it calls p.setAllAncestorAtFileNodesDirty().

Usually, code *should* this setter, despite its cost, because it update's Leo's outline pane properly. Calling c.redraw() is *not* enough.

setHeadString(*s*)

setIcon()

setMarked()

setOrphan()

setSelected()

setSelection(*start*, *length*)

setTnodeText(*s*)

setVisited()

simpleLevel()
Return the number of p's parents.

sort_key(*p*)

status()

subtree()
Yield all positions in p's subtree, but not p.

subtree_iter()
Yield all positions in p's subtree, but not p.

subtree_with_unique_tnodes_iter()
Yield p and all other unique positions in p's subtree.

subtree_with_unique_vnodes_iter()
Yield p and all other unique positions in p's subtree.

textOffset()
Return the fcol offset of self. Return None if p is has no ancestor @<file> node. http://tinyurl.com/5nescw

threadBack()

threadNext()

tnodes_iter()
Yield p.v and all vnodes in p's subtree.

u
p.u property

unique_nodes()
Yield p.v and all unique vnodes in p's subtree.

unique_subtree()
Yield p and all other unique positions in p's subtree.

unique_tnodes_iter()
Yield p.v and all unique vnodes in p's subtree.

unique_vnodes_iter()
Yield p.v and all unique vnodes in p's subtree.

validateOutlineWithParent(*pv*)

visBack(*c*)

visNext(*c*)

vnodes_iter()
Yield p.v and all vnodes in p's subtree.

leo.core.leoNodes.**Poslist**
alias of *leo.core.leoNodes.PosList*

leo.core.leoNodes.**VNode**
alias of *leo.core.leoNodes.VNodeBase*

class leo.core.leoNodes.**VNodeBase**(*context*, *gnx=None*)
Bases: object

anyAtFileNodeName()
Return the file name following an @file node or an empty string.

atAsisFileNodeName()

atAutoNodeName(*h=None*)

atAutoRstNodeName(*h=None*)

atCleanNodeName()

atEditNodeName()

atFileNodeName()

atNoSentFileNodeName()

atNoSentinelsFileNodeName()

atRstFileNodeName()

atShadowFileNodeName()

atSilentFileNodeName()

atThinFileNodeName()

b
VNode body string property

bodyString()

body_unicode_warning = False

childrenModified()

cleanHeadString()

clearClonedBit()

clearDirty()
Clear the vnode dirty bit.

clearMarked()

clearOrphan()

clearVisited()

clearWriteBit()

cloneAsNthChild(*parent_v*, *n*)

clonedBit = 1

computeIcon()

contentModified()

contract()
Contract the node.

copyTree(*copyMarked=False*)
Return an all-new tree of vnodes that are copies of self and all its descendants.

Important: the v.parents ivar must be [] for all nodes. v._addParentLinks will set all parents.

directParents()
(New in 4.2) Return a list of all direct parent vnodes of a VNode.

This is NOT the same as the list of ancestors of the VNode.

dirtyBit = 512

dump(*label=''*)

dumpLink(*link*)

expand()
Expand the node.

expandedBit = 4

findAllPotentiallyDirtyNodes()

findAtFileName(*names*, *h=''*)
Return the name following one of the names in nameList or ""

firstChild()

getBody()

gnx
VNode gnx property

h
VNode headline string property

hasBody()
Return True if this VNode contains body text.

hasChildren()

hasFirstChild()

headString()
Return the headline string.

head_unicode_warning = False

initBodyString(*s*)

initClonedBit(*val*)

initExpandedBit()
Init self.statusBits.

initHeadString(*s*)

initMarkedBit()

initStatus(*status*)

insertAsFirstChild()

insertAsLastChild()

insertAsNthChild(*n*)

isAnyAtFileNode()
Return True if v is any kind of @file or related node.

isAtAllNode()
Returns True if the receiver contains @others in its body at the start of a line.

isAtAsisFileNode()

isAtAutoNode()

isAtAutoRstNode()

isAtCleanNode()

isAtEditNode()

isAtFileNode()

isAtIgnoreNode()
Returns True if the receiver contains @ignore in its body at the start of a line.

or if the headline starts with @ignore.

isAtNoSentFileNode()

isAtNoSentinelsFileNode()

isAtOthersNode()
Returns True if the receiver contains @others in its body at the start of a line.

isAtRstFileNode()

isAtShadowFileNode()

isAtSilentFileNode()

isAtThinFileNode()

isCloned()

isDirty()

isExpanded()
Return True if the VNode expansion bit is set.

isMarked()

isNthChildOf(*n*, *parent_v*)
Return True if v is the n'th child of parent_v.

isOrphan()

isSelected()

isTopBitSet()

isVisited()

isWriteBit()

lastChild()

markedBit = 8

matchHeadline(*pattern*)
Returns True if the headline matches the pattern ignoring whitespace and case.

The headline may contain characters following the successfully matched pattern.

nthChild(*n*)

numberOfChildren()

orphanBit = 2048

restoreCursorAndScroll()
Restore the cursor position and scroll so it is visible.

richTextBit = 128

saveCursorAndScroll()

selectedBit = 32

setAllAncestorAtFileNodesDirty()

setBodyString(*s*)

setClonedBit()

setDirty()
Set the vnode dirty bit.

setHeadString(*s*)

setHeadText(*s*)

setIcon()

setMarked()

setOrphan()
Set the vnode's orphan bit.

setSelected()

setSelection(*start*, *length*)

setTnodeText(*s*)

setVisited()

setWriteBit()

status()

topBit = 64

u
VNode u property

unicode_warning_given = False

visitedBit = 256

writeBit = 1024

leo.core.leoNodes.**position**
alias of *leo.core.leoNodes.Position*

leo.core.leoNodes.**vnode**
alias of *leo.core.leoNodes.VNodeBase*

1.2.1.28 leoPlugins Module

Classes relating to Leo's plugin architecture.

class leo.core.leoPlugins.**BaseLeoPlugin**(*tag*, *keywords*)

Bases: object

A Convenience class to simplify plugin authoring

- import the base class:

```
from leoPlugins import leo.core.leoBasePlugin as leoBasePlugin
```

- create a class which inherits from leoBasePlugin:

```
class myPlugin(leoBasePlugin):
```

- in the __init__ method of the class, call the parent constructor:

```
def __init__(self, tag, keywords):
    leoBasePlugin.__init__(self, tag, keywords)
```

- put the actual plugin code into a method; for this example, the work is done by myPlugin.handler()
- **put the class in a file which lives in the <LeoDir>/plugins directory** for this example it is named myPlugin.py
- add code to register the plugin:

```
leoPlugins.registerHandler("after-create-leo-frame", Hello)
```

BaseLeoPlugins has 3 *methods* for setting commands

- setCommand:

```
def setCommand(self, commandName, handler,
        shortcut = None, pane = 'all', verbose = True):
```

- setMenuItem:

```
def setMenuItem(self, menu, commandName = None, handler = None):
```

- setButton:

```
def setButton(self, buttonText = None, commandName = None, color = None):
```

variables

CommandName the string typed into minibuffer to execute the handler

Handler the method in the class which actually does the work

Shortcut the key combination to activate the command

Menu a string designating on of the menus ('File', Edit', 'Outline', ...)

ButtonText the text to put on the button if one is being created.

Contents of file <LeoDir>/plugins/hello.py:

```
class Hello(BaseLeoPlugin):
    def __init__(self, tag, keywords):

        # call parent __init__
        BaseLeoPlugin.__init__(self, tag, keywords)

        # if the plugin object defines only one command,
        # just give it a name. You can then create a button and menu entry
        self.setCommand('Hello', self.hello)
        self.setButton()
        self.setMenuItem('Cmds')

        # create a command with a shortcut
        self.setCommand('Hola', self.hola, 'Alt-Ctrl-H')

        # create a button using different text than commandName
        self.setButton('Hello in Spanish')

        # create a menu item with default text
        self.setMenuItem('Cmds')

        # define a command using setMenuItem
        self.setMenuItem('Cmds', 'Ciao baby', self.ciao)

    def hello(self, event):
        g.pr("hello from node %s" % self.c.p.h)

    def hola(self, event):
        g.pr("hola from node %s" % self.c.p.h)

    def ciao(self, event):
        g.pr("ciao baby (%s)" % self.c.p.h)

leoPlugins.registerHandler("after-create-leo-frame", Hello)
```

setButton(*buttonText=None*, *commandName=None*, *color=None*)
Associate an existing command with a 'button'

setCommand(*commandName*, *handler*, *shortcut=''*, *pane='all'*, *verbose=True*)
Associate a command name with handler code, optionally defining a keystroke shortcut

setMenuItem(*menu*, *commandName=None*, *handler=None*)
Create a menu item in 'menu' using text 'commandName' calling handler 'handler' if commandName and handler are none, use the most recently defined values

class leo.core.leoPlugins.**CommandChainDispatcher**(*commands=None*)
Bases: object

Dispatch calls to a chain of commands until some func can handle it

Usage: instantiate, execute "add" to add commands (with optional priority), execute normally via f() calling mechanism.

add(*func*, *priority=0*)
Add a func to the cmd chain with given priority

class leo.core.leoPlugins.**LeoPluginsController**
Bases: object

The global plugins controller, g.app.pluginsController

callTagHandler(*bunch*, *tag*, *keywords*)
Call the event handler.

doHandlersForTag(*tag*, *keywords*)
Execute all handlers for a given tag, in alphabetical order. The caller, doHook, catches all exceptions.

doPlugins(*tag*, *keywords*)
The default g.app.hookFunction.

finishCreate()

getHandlersForOneTag(*tag*)

getHandlersForTag(*tags*)

getLoadedPlugins()

getPluginModule(*moduleName*)

isLoaded(*fn*)

loadHandlers(*tag*, *keys*)
Load all enabled plugins.

Using a module name (without the trailing .py) allows a plugin to be loaded from outside the leo/plugins directory.

loadOnePlugin(*moduleOrFileName*, *tag='open0'*, *verbose=False*)
Load one plugin from a file name or module. Use extensive tracing if –trace-plugins is in effect.

Using a module name allows plugins to be loaded from outside the leo/plugins directory.

on_idle()
Call all idle-time hooks.

plugin_signon(*module_name*, *verbose=False*)
Print the plugin signon.

printHandlers(*c*, *moduleName=None*)
Print the handlers for each plugin.

printPlugins(*c*)
Print all enabled plugins.

printPluginsInfo(*c*)
Print the file name responsible for loading a plugin.

This is the first .leo file containing an @enabled-plugins node that enables the plugin.

registerExclusiveHandler(*tags*, *fn*)
Register one or more exclusive handlers

registerHandler(*tags*, *fn*)
Register one or more handlers

registerOneExclusiveHandler(*tag*, *fn*)
Register one exclusive handler

registerOneHandler(*tag*, *fn*)
Register one handler

regularizeName(*fn*)
Return the name used as a key to this modules dictionaries.

reloadSettings()

setLoaded(*fn*, *m*)

unloadOnePlugin(*moduleOrFileName*, *verbose=False*)

unregisterHandler(*tags*, *fn*)

unregisterOneHandler(*tag*, *fn*)

exception leo.core.leoPlugins.**TryNext**(**args*, ***kwargs*)
Bases: exceptions.Exception

Try next hook exception.

Raise this in your hook function to indicate that the next hook handler should be used to handle the operation. If you pass arguments to the constructor those arguments will be used by the next hook instead of the original ones.

leo.core.leoPlugins.**init**()
Init g.app.pluginsController.

leo.core.leoPlugins.**registerHandler**(*tags*, *fn*)
A wrapper so plugins can still call leoPlugins.registerHandler.

1.2.1.29 leoPymacs Module

A module to allow the Pymacs bridge to access Leo data.

All code in this module must be called *from* Emacs: calling Pymacs.lisp in other situations will hang Leo.

Notes:

- The init method adds the parent directory of leoPymacs.py to Python's sys.path. This is essential to make imports work from inside Emacs.
- As of Leo 4.5, the following code, when executed from an Emacs buffer, will open trunk/leo/test.leo:

```
(pymacs-load "c:\leo.repo\trunk\leo\core\leoPymacs" "leo-")
(setq c (leo-open "c:\leo.repo\trunk\leo\test\test.leo"))
```

Note that full path names are required in each case.

leo.core.leoPymacs.**dump**(*anObject*)

leo.core.leoPymacs.**get_app**()
Scripts can use g.app.scriptDict for communication with pymacs.

leo.core.leoPymacs.**get_g**()

leo.core.leoPymacs.**hello**()

leo.core.leoPymacs.**init**()

leo.core.leoPymacs.**open**(*fileName=None*)

leo.core.leoPymacs.**run_script**(*c*, *script*, *p=None*)

leo.core.leoPymacs.**script_result**()

1.2.1.30 leoRst Module

1.2.1.31 leoSessions Module

Support for sessions in Leo.

exception leo.core.leoSessions.**LeoSessionException**
Bases: exceptions.Exception

class leo.core.leoSessions.**SessionManager**
Bases: object

clear_session(*c*)
Close all tabs except the presently selected tab.

get_session()
Return a list of UNLs for open tabs.

get_session_path()
Return the path to the session file.

load_session(*c=None*, *unls=None*)
Open a tab for each item in UNLs & select the indicated node in each.

load_snapshot()
Load a snapshot of a session from the leo.session file.

Called when –restore-session is in effect.

save_snapshot(*c=None*)
Save a snapshot of the present session to the leo.session file.

Called automatically during shutdown when –session-save is in effect.

leo.core.leoSessions.**session_clear_command**(*event*)
Close all tabs except the presently selected tab.

leo.core.leoSessions.**session_create_command**(*event*)
Create a new @session node.

leo.core.leoSessions.**session_refresh_command**(*event*)
Refresh the current @session node.

leo.core.leoSessions.**session_restore_command**(*event*)
Open a tab for each item in the @session node & select the indicated node in each.

leo.core.leoSessions.**session_snapshot_load_command**(*event*)
Load a snapshot of a session from the leo.session file.

leo.core.leoSessions.**session_snapshot_save_command**(*event*)
Save a snapshot of the present session to the leo.session file.

1.2.1.32 leoShadow Module

leoShadow.py

This code allows users to use Leo with files which contain no sentinels and still have information flow in both directions between outlines and derived files.

Private files contain sentinels: they live in the Leo-shadow subdirectory. Public files contain no sentinels: they live in the parent (main) directory.

When Leo first reads an @shadow we create a file without sentinels in the regular directory.

The slightly hard thing to do is to pick up changes from the file without sentinels, and put them into the file with sentinels.

Settings: - @string shadow_subdir (default: .leo_shadow): name of the shadow directory.

- @string shadow_prefix (default: x): prefix of shadow files. This prefix allows the shadow file and the original file to have different names. This is useful for name-based tools like py.test.

class leo.core.leoShadow.**ShadowController**(*c*, *trace=False*, *trace_writers=False*)
Bases: object

A class to manage @shadow files

class AtShadowTestCase(*c*, *p*, *shadowController*, *delims=None*, *trace=False*)
Bases: unittest.case.TestCase

Support @shadow-test nodes.

These nodes should have two descendant nodes: 'before' and 'after'.

createSentinelNode(*root*, *p*)
Write p's tree to a string, as if to a file.

fail(*msg=None*)
Mark an AtShadowTestCase as having failed.

findNode(*c*, *p*, *headline*)
Return the node in p's subtree with given headline.

makePrivateLines(*p*)
Return a list of the lines of p containing sentinels.

makePublicLines(*lines*)
Return the public lines in lines.

mungePrivateLines(*lines*, *find*, *replace*)
Change the 'find' the 'replace' pattern in sentinel lines.

runTest(*define_g=True*)
AtShadowTestCase.runTest.

setUp()
AtShadowTestCase.setup.

shortDescription()
AtShadowTestCase.shortDescription.

tearDown()
AtShadowTestCase.tearDown.

class Marker(*delims*)
Bases: object

A class representing comment delims in @shadow files.

getDelims()
Return the pair of delims to be used in sentinel lines.

isSentinel(*s*, *suffix=''*)
Return True is line s contains a valid sentinel comment.

isVerbatimSentinel(*s*)
Return True if s is an @verbatim sentinel.

baseDirName()

check_output()
Check that we produced a valid output.

dirName (*filename*)
Return the directory for filename.

dump_args ()
Dump the argument lines.

dump_lines (*lines*, *title*)
Dump the given lines.

error (*s*, *silent=False*)

findLeoLine (*lines*)
Return the @+leo line, or ''.

init_data ()
Init x.sentinels and x.trailing_sentinels arrays. Return the list of non-sentinel lines in x.old_sent_lines.

init_ivars (*new_public_lines*, *old_private_lines*, *marker*)
Init all ivars used by propagate_changed_lines & its helpers.

isSignificantPublicFile (*fn*)
This tells the AtFile.read logic whether to import a public file or use an existing public file.

makeShadowDirectory (*fn*)
Make a shadow directory for the **public** fn.

markerFromFileLines (*lines*, *fn*)
Return the sentinel delimiter comment to be used for filename.

markerFromFileName (*filename*)
Return the sentinel delimiter comment to be used for filename.

message (*s*)

op_bad (*tag*, *ai*, *aj*, *bi*, *bj*)
Report an unexpected opcode.

op_delete (*tag*, *ai*, *aj*, *bi*, *bj*)
Handle the 'delete' opcode.

op_equal (*tag*, *ai*, *aj*, *bi*, *bj*)
Handle the 'equal' opcode.

op_insert (*tag*, *ai*, *aj*, *bi*, *bj*)
Handle the 'insert' opcode.

op_replace (*tag*, *ai*, *aj*, *bi*, *bj*)
Handle the 'replace' opcode.

pathName (*filename*)
Return the full path name of filename.

preprocess (*lines*)
Preprocess public lines, adding newlines as needed. This happens before the diff.

propagate_changed_lines (*new_public_lines*, *old_private_lines*, *marker*, *p=None*)
The Mulder update algorithm, revised by EKR.

Use the diff between the old and new public lines to insperse sentinels from old_private_lines into the result.

The algorithm never deletes or rearranges sentinels. However, verbatim sentinels may be inserted or deleted as needed.

propagate_changes(*old_public_file*, *old_private_file*)
Propagate the changes from the public file (without_sentinels) to the private file (with_sentinels)

put_plain_line(*line*)
Put a plain line to x.results, inserting verbatim lines if necessary.

put_sentinels(*i*)
Put all the sentinels to the results

reloadSettings()
ShadowController.reloadSettings.

replaceFileWithString(*fn*, *s*)
Replace the file with s if s is different from theFile's contents.

Return True if theFile was changed.

separate_sentinels(*lines*, *marker*)
Separates regular lines from sentinel lines. Do not return @verbatim sentinels.

Returns (regular_lines, sentinel_lines)

shadowDirName(*filename*)
Return the directory for the shadow file corresponding to filename.

shadowPathName(*filename*)
Return the full path name of filename, resolved using c.fileName()

show_error(*lines1*, *lines2*, *message*, *lines1_message*, *lines2_message*)

show_error_lines(*lines*, *fileName*)

unlink(*filename*, *silent=False*)
Unlink filename from the file system. Give an error on failure.

updatePublicAndPrivateFiles(*root*, *fn*, *shadow_fn*)
handle crucial @shadow read logic.

This will be called only if the public and private files both exist.

verbatim_error()

1.2.1.33 leoTangle Module

Support for @root and Leo's tangle and untangle commands.

Everything in this file is deprecated, but will remain "forever".

class leo.core.leoTangle.**BaseTangleCommands**(*c*)
Bases: object

The base class for Leo's tangle and untangle commands.

class RegexpForLanguageOrComment
Bases: object

re = <module 're' from '/home/docs/checkouts/readthedocs.org/user_builds/leo-editor

regex = <_sre.SRE_Pattern object>

cleanup()

compare_comments(*s1*, *s2*)

compare_section_names(*s1*, *s2*)

copy(*s*)

error(*s*)

forgiving_compare(*name*, *part*, *s1*, *s2*)

handle_newline(*s*, *i*, *delims*)

initTangleCommand()

initUntangleCommand()

init_directive_ivars()

init_ivars()

is_end_of_directive(*s*, *i*)

is_end_of_string(*s*, *i*, *delim*)

is_escaped(*s*, *i*)

is_section_name(*s*, *i*)

is_sentinel_line(*s*, *i*)

is_sentinel_line_with_data(*s*, *i*)

mismatch(*message*)

oblank()

oblanks(*n*)

onl()

os(*s*)

otab()

otabs(*n*)

parent_language_comment_settings(*p*, *lang_dict*)

pathError(*s*)

push_new_DefNode(*name*, *indent*, *part*, *of*, *nl_flag*)

push_parts(*reflist*)

put_PartNode(*part*, *no_first_lws_flag*)

put_all_roots()

put_code(*s*, *no_first_lws_flag*, *delims*)

put_doc(*s*, *delims*)

put_leading_ws(*n*)

put_newline(*s*, *i*, *no_first_lws_flag*)

put_section(*s*, *i*, *name*, *name_end*, *delims*)

refpart_stack_dump()

reloadSettings()

reload_settings()

scanAllDirectives(*p*)
Scan VNode p and p's ancestors looking for directives, setting corresponding tangle ivars and globals.

scan_derived_file(*s*)

scan_short_val(*s*, *i*)

section_check(*name*)

select_next_sentinel(*part_start_flag=True*)
The next sentinel will be either (a) a section part reference, using the "before" comment style for that part - when there are section references yet to interpolate for this part - when we're followed by another part for this section (b) an end sentinel using the "after" comment style for the current part - when we've exhausted the parts for this section or (c) end of file for the root section The above requires that the parts in the tst be aware of the section interpolations each part will make

setRootFromHeadline(*p*)

setRootFromText(*s*, *report_errors=True*)

skip_body(*p*, *delims*)
The following subsections contain the interface between the Tangle and Untangle commands. This interface is an important hack, and allows Untangle to avoid duplicating the logic in skip_tree and its allies.

The aha is this: just at the time the Tangle command enters a definition into the symbol table, all the information is present that Untangle needs to update that definition.

To get whitespace exactly right we retain the outline's leading whitespace and remove leading whitespace from the updated definition.

skip_code(*s*, *i*, *delims*)

skip_doc(*s*, *i*, *delims*)

skip_headline(*p*)

skip_section_name(*s*, *i*)

st_check()
Checks the given symbol table for defined but never referenced sections.

st_dump(*verbose_flag=True*)

st_dump_node(*section*)

st_enter(*name*, *code*, *doc*, *delims_begin*, *delims_end*, *is_root_flag=False*)
Enters names and their associated code and doc parts into the given symbol table.

st_enter_root_name(*name*, *code*, *doc*, *delims_begin*, *delims_end*)

st_enter_section_name(*name*, *code*, *doc*, *delims_begin*, *delims_end*)
Enters a section name into the given symbol table.

The code and doc pointers are None for references.

st_lookup(*name*, *is_root_flag=False*)
Looks up name in the symbol table and creates a TstNode for it if it does not exist.

standardize_name(*name*)
Removes leading and trailing brackets, converts white space to a single blank and converts to lower case.

tangle(*event=None*, *p=None*)

tangleAll(*event=None*)

tangleMarked(*event=None*)

tanglePass1(*p_in*, *delims*)
The main routine of tangle pass 1

tanglePass2()

tangleTree(*p*, *report_errors*)
Tangles all nodes in the tree whose root is p.

Reports on its results if report_errors is True.

token_type(*s*, *i*, *report_errors=True*)
This method returns a code indicating the apparent kind of token at the position i.

The caller must determine whether section definiton tokens are valid.

returns (kind, end) and sets global root_name using setRootFromText(). end is only valid for kind in (section_ref, section_def, at_root).

untangle(*event=None*, *p=None*)

untangleAll(*event=None*)

untangleMarked(*event=None*)

untangleRoot(*root*, *begin*, *end*)

untangleTree(*p*, *report_errors*)

update_current_vnode(*s*)
Called from within the Untangle logic to update the body text of self.p.

update_def(*name*, *part_number*, *head*, *code*, *tail*, *is_root_flag=False*)

ust_dump()

ust_enter(*name*, *part*, *of*, *code*, *nl_flag*, *is_root_flag=False*)
This routine enters names and their code parts into the given table. The 'part' and 'of' parameters are taken from the "(part n of m)" portion of the line that introduces the section definition in the C code.

If no part numbers are given the caller should set the 'part' and 'of' parameters to zero. The caller is reponsible for checking for duplicate parts.

This function handles names scanned from a source file; the corresponding st_enter routine handles names scanned from outlines.

ust_lookup(*name*, *part_number*, *is_root_flag=False*, *update_flag=False*)
Search the given table for a part matching the name and part number.

ust_warn_about_orphans()
Issues a warning about any sections in the derived file for which no corresponding section has been seen in the outline.

warning(*s*)

class leo.core.leoTangle.**DefNode**(*name*, *indent*, *part*, *of*, *nl_flag*, *code*)
Bases: object

class leo.core.leoTangle.**PartNode**(*name*, *code*, *doc*, *is_root*, *is_dirty*, *delims*)
Bases: object

reflist(*refs=False*)

class leo.core.leoTangle.**RootAttributes**(*tangle_state*)
Bases: object

class leo.core.leoTangle.**TangleCommands**(*c*)
Bases: *leo.core.leoTangle.BaseTangleCommands*

A class that implements Leo' tangle and untangle commands.

class leo.core.leoTangle.**TstNode**(*name*, *root_flag*)
Bases: object

dump()

class leo.core.leoTangle.**UstNode**(*name*, *code*, *part*, *of*, *nl_flag*, *update_flag*)
Bases: object

dump()

1.2.1.34 leoTest Module

1.2.1.35 leoUndo Module

Leo's undo/redo manager.

class leo.core.leoUndo.**Undoer**(*c*)
Bases: object

A class that implements unlimited undo and redo.

afterChangeGroup(*p*, *undoType*, *reportFlag=False*, *dirtyVnodeList=None*)
Create an undo node for general tree operations using d created by beforeChangeGroup

afterChangeNodeContents(*p*, *command*, *bunch*, *dirtyVnodeList=None*, *inHead=False*)
Create an undo node using d created by beforeChangeNode.

afterChangeTree(*p*, *command*, *bunch*)
Create an undo node for general tree operations using d created by beforeChangeTree

afterClearRecentFiles(*bunch*)

afterCloneMarkedNodes(*p*)

afterCloneNode(*p*, *command*, *bunch*, *dirtyVnodeList=None*)

afterCopyMarkedNodes(*p*)

afterDehoist(*p*, *command*)

afterDeleteMarkedNodes(*data*, *p*)

afterDeleteNode(*p*, *command*, *bunch*, *dirtyVnodeList=None*)

afterDemote(*p*, *followingSibs*, *dirtyVnodeList*)
Create an undo node for demote operations.

afterHoist(*p*, *command*)

afterInsertNode(*p*, *command*, *bunch*, *dirtyVnodeList=None*)

afterMark(*p*, *command*, *bunch*, *dirtyVnodeList=None*)
Create an undo node for mark and unmark commands.

afterMoveNode(*p*, *command*, *bunch*, *dirtyVnodeList=None*)

afterPromote(*p*, *children*, *dirtyVnodeList*)
Create an undo node for demote operations.

afterSort(*p*, *bunch*, *dirtyVnodeList*)
Create an undo node for sort operations

beforeChangeGroup(*p*, *command*, *verboseUndoGroup=True*)
Prepare to undo a group of undoable operations.

beforeChangeNodeContents(*p*, *oldBody=None*, *oldHead=None*, *oldYScroll=None*)
Return data that gets passed to afterChangeNode

beforeChangeTree(*p*)

beforeClearRecentFiles()

beforeCloneNode(*p*)

beforeDeleteNode(*p*)

beforeInsertNode(*p*, *pasteAsClone=False*, *copiedBunchList=None*)

beforeMark(*p*, *command*)

beforeMoveNode(*p*)

beforeSort(*p*, *undoType*, *oldChildren*, *newChildren*, *sortChildren*)
Create an undo node for sort operations.

canRedo()

canUndo()

clearOptionalIvars()

clearUndoState()
Clears then entire Undo state.

All non-undoable commands should call this method.

cmd()
Command decorator for the Undoer class.

createCommonBunch(*p*)
Return a bunch containing all common undo info. This is mostly the info for recreating an empty node at position p.

createTnodeUndoInfo(*v*)
Create a bunch containing all info needed to recreate a VNode.

createVnodeUndoInfo(*v*)
Create a bunch containing all info needed to recreate a VNode for undo.

cutStack()

dumpBead(*n*)

dumpTopBead()

enableMenuItems()

getBead(*n*)
Set Undoer ivars from the bunch at the top of the undo stack.

onSelect(*old_p*, *p*)

peekBead(*n*)

pushBead(*bunch*)

putIvarsToVnode(*p*)

recognizeStartOfTypingWord(*old_lines*, *old_row*, *old_col*, *old_ch*, *new_lines*, *new_row*, *new_col*, *new_ch*, *prev_row*, *prev_col*)
A potentially user-modifiable method that should return True if the typing indicated by the params starts a new 'word' for the purposes of undo with 'word' granularity.

u.setUndoTypingParams calls this method only when the typing could possibly continue a previous word. In other words, undo will work safely regardless of the value returned here.

old_ch is the char at the given (Tk) row, col of old_lines. new_ch is the char at the given (Tk) row, col of new_lines.

The present code uses only old_ch and new_ch. The other arguments are given for use by more sophisticated algorithms.

redo(*event=None*)
Redo the operation undone by the last undo.

redoClearRecentFiles()

redoCloneMarkedNodes()

redoCloneNode()

redoCopyMarkedNodes()

redoDehoistNode()

redoDeleteMarkedNodes()

redoDeleteNode()

redoDemote()

redoGroup()
Process beads until the matching 'afterGroup' bead is seen.

redoHelper()

redoHoistNode()

redoInsertNode()

redoMark()

redoMenuName(*name*)

redoMove()

redoNodeContents()

redoPromote()

redoSort()

redoTree()
Redo replacement of an entire tree.

redoTyping()

reloadSettings()
Undoer.reloadSettings.

restoreTnodeUndoInfo(*bunch*)

restoreTree(*treeInfo*)
Use the tree info to restore all VNode data, including all links.

restoreVnodeUndoInfo(*bunch*)
Restore all ivars saved in the bunch.

saveTree(*p*, *treeInfo=None*)
Return a list of tuples with all info needed to handle a general undo operation.

setIvarsFromBunch(*bunch*)

setIvarsFromVnode(*p*)

setRedoType(*theType*)

setUndoType(*theType*)

setUndoTypes()

setUndoTypingParams(*p*, *undo_type*, *oldText*, *newText*, *oldSel=None*, *newSel=None*, *oldYview=None*)
Save enough information to undo or redo typing operation.

Do nothing when called from the undo/redo logic because the Undo and Redo commands merely reset the bead pointer.

trace()

undo(*event=None*)
Undo the operation described by the undo parameters.

undoClearRecentFiles()

undoCloneMarkedNodes()

undoCloneNode()

undoCopyMarkedNodes()

undoDehoistNode()

undoDeleteMarkedNodes()

undoDeleteNode()

undoDemote()

undoGroup()
Process beads until the matching 'beforeGroup' bead is seen.

undoHelper()

undoHoistNode()

undoInsertNode()

undoMark()

undoMenuName(*name*)

undoMove()

undoNodeContents()
Undo all changes to the contents of a node, including headline and body text, and marked bits.

undoPromote()

undoRedoText(*p*, *leading*, *trailing*, *oldMidLines*, *newMidLines*, *oldNewlines*, *newNewlines*, *tag='undo'*, *undoType=None*)
Handle text undo and redo: converts _new_ text into _old_ text.

undoRedoTree(*p*, *new_data*, *old_data*)
Replace p and its subtree using old_data during undo.

undoSort()

undoTree()
Redo replacement of an entire tree.

undoTyping()

updateMarks(*oldOrNew*)
Update dirty and marked bits.

1.2.1.36 leoVersion Module

A module holding the following version-related info:

leoVersion.branch: The git branch name, or ‘’. leoVersion.build: The timestamp field from commit_timestamp.json. leoVersion.date: The asctime field from commit_timestamp.json. leoVersion.version: Leo’s version number, set below.

1.2.1.37 runLeo Module

1.2.2 extensions Package

1.2.2.1 extensions Package

1.2.2.2 asciidoc Module

asciidoc - converts an AsciiDoc text file to HTML or DocBook

Copyright (C) 2002-2010 Stuart Rackham. Free use of this software is granted under the terms of the GNU General Public License (GPL).

class leo.extensions.asciidoc.**AbstractBlock**

blocknames = []

dump()
Write block definition to stdout.

error(*msg*, *cursor=None*, *halt=False*)

get_param(*name*, *params=None*)
Return named processing parameter from params dictionary. If the parameter is not in params look in self.parameters.

get_subs(*params=None*)
Return (presubs,postsubs) tuple.

is_conf_entry(*param*)
Return True if param matches an allowed configuration file entry name.

isnext()
Check if this block is next in document reader.

load(*defname*, *entries*)
Update block definition from section ‘entries’ dictionary.

merge_attributes(*attrs*, *params=[]*)
Use the current block's attribute list (attrs dictionary) to build a dictionary of block processing parameters (self.parameters) and tag substitution attributes (self.attributes).

1. Copy the default parameters (self.*) to self.parameters. self.parameters are used internally to render the current block. Optional params array of additional parameters.

2. Copy attrs to self.attributes. self.attributes are used for template and tag substitution in the current block.

3. If a style attribute was specified update self.parameters with the corresponding style parameters; if there are any style parameters remaining add them to self.attributes (existing attribute list entries take precedence).

4. Set named positional attributes in self.attributes if self.posattrs was specified.

5. Finally self.parameters is updated with any corresponding parameters specified in attrs.

pop_blockname()
On block exits restore previous (parent) 'blockname' attribute or undefine it if we're no longer inside a block.

push_blockname(*blockname=None*)
On block entry set the 'blockname' attribute. Only applies to delimited blocks, lists and tables.

short_name()
Return the text following the first dash in the section name.

translate()
Translate block from document reader.

update_parameters(*src*, *dst=None*, *all=False*)
Parse processing parameters from src dictionary to dst object. dst defaults to self.parameters. If all is True then copy src entries that aren't parameter names.

validate()
Validate block after the complete configuration has been loaded.

class leo.extensions.asciidoc.**AbstractBlocks**
List of block definitions.

BLOCK_TYPE = None

PREFIX = ''

dump()

isnext()

load(*sections*)
Load block definition from 'sections' dictionary.

validate()
Validate the block definitions.

class leo.extensions.asciidoc.**AttrDict**
Bases: dict

Like a dictionary except values can be accessed as attributes i.e. obj.foo can be used in addition to obj['foo']. If an item is not present None is returned.

class leo.extensions.asciidoc.**AttributeEntry**
Static methods and attributes only.

attributes = {}

static isnext()

name = None

name2 = None

pattern = None

subs = None

static translate()

value = None

class leo.extensions.asciidoc.**AttributeList**
Static methods and attributes only.

attrs = {}

static consume()
Add attribute list to the dictionary 'd' and reset the list.

static initialize()

static isnext()

match = None

pattern = None

static style()

static subs()
Substitute single quoted attribute values normally.

static translate()

class leo.extensions.asciidoc.**BlockTitle**
Static methods and attributes only.

static consume()
If there is a title add it to dictionary 'd' then reset title.

static isnext()

pattern = None

title = None

static translate()

class leo.extensions.asciidoc.**CalloutMap**

add(*listindex*)

static calloutid(*calloutindex*)

calloutids(*listindex*)

listclose()

validate(*maxlistindex*)

class leo.extensions.asciidoc.**Cell**(*data*, *span_spec=None*, *align_spec=None*, *style=None*)

clone_reserve()
Return a clone of self to reserve vertically spanned cell.

class leo.extensions.asciidoc.**Column**(*width=None*, *align_spec=None*, *style=None*)
Table column.

class leo.extensions.asciidoc.**Column_OLD**
Table column.

class leo.extensions.asciidoc.**Config**
Methods to process configuration files.

ENTRIES_SECTIONS = ('tags', 'miscellaneous', 'attributes', 'specialcharacters', 'specia

dump()
Dump configuration to stdout.

entries_section(*section_name*)
Return True if conf file section contains entries, not a markup template.

expand_all_templates()

expand_templates(*entries*)
Expand any template::[] macros in a list of section entries.

find_config_dir(**dirnames*)
Return path of configuration directory. Try all the well known locations. Return None if directory not found.

find_in_dirs(*filename*, *dirs=None*)
Find conf files from dirs list. Return list of found file paths. Return empty list if not found in any of the locations.

get_load_dirs()
Return list of well known paths with conf files.

init(*cmd*)
Check Python version and locate the executable and configuration files directory. cmd is the asciidoc command or asciidoc.py path.

load_backend(*dirs=None*)
Load the backend configuration files from dirs list. If dirs not specified try all the well known locations. If a <backend>.conf file was found return it's full path name, if not found return None.

load_file(*fname*, *dir=None*, *include=[]*, *exclude=[]*)
Loads sections dictionary with sections from file fname. Existing sections are overlaid. The 'include' list contains the section names to be loaded. The 'exclude' list contains section names not to be loaded. Return False if no file was found in any of the locations.

load_filters(*dirs=None*)
Load filter configuration files from 'filters' directory in dirs list. If dirs not specified try all the well known locations. Suppress loading if a file named __noautoload__ is in same directory as the conf file unless the filter has been specified with the –filter command-line option (in which case it is loaded unconditionally).

load_from_dirs(*filename*, *dirs=None*, *include=[]*)
Load conf file from dirs list. If dirs not specified try all the well known locations. Return False if no file was sucessfully loaded.

load_miscellaneous(*d*)
Set miscellaneous configuration entries from dictionary 'd'.

load_sections(*sections*, *attrs=None*)
Loads sections dictionary. Each dictionary entry contains a list of lines. Updates 'attrs' with parsed [attributes] section entries.

parse_replacements(*sect='replacements'*)
Parse replacements section into self.replacements dictionary.

parse_specialsections()
Parse specialsections section to self.specialsections dictionary.

parse_specialwords()
Parse special words section into self.specialwords dictionary.

parse_tags()
Parse [tags] section entries into self.tags dictionary.

section2tags(*section*, *d={}*, *skipstart=False*, *skipend=False*)
Perform attribute substitution on 'section' using document attributes plus 'd' attributes. Return tuple (stag,etag) containing pre and post | placeholder tags. 'skipstart' and 'skipend' are used to suppress substitution.

static set_replacement(*rep*, *replacements*)
Add pattern and replacement to replacements dictionary.

set_theme_attributes()

subs_replacements(*s*, *sect='replacements'*)
Substitute patterns from self.replacements in 's'.

subs_section(*section*, *d*)
Section attribute substitution using attributes from document.attributes and 'd'. Lines containing undefinded attributes are deleted.

subs_specialchars(*s*)
Perform special character substitution on string 's'.

subs_specialchars_reverse(*s*)
Perform reverse special character substitution on string 's'.

subs_specialwords(*s*)
Search for word patterns from self.specialwords in 's' and substitute using corresponding macro.

tag(*name*, *d=None*)
Returns (starttag,endtag) tuple named name from configuration file [tags] section. Raise error if not found. If a dictionary 'd' is passed then merge with document attributes and perform attribute substitution on tags.

validate()
Check the configuration for internal consistancy. Called after all configuration files have been loaded.

class leo.extensions.asciidoc.**DelimitedBlock**
Bases: *leo.extensions.asciidoc.AbstractBlock*

dump()

isnext()

load(*name*, *entries*)

translate()

class leo.extensions.asciidoc.**DelimitedBlocks**
Bases: *leo.extensions.asciidoc.AbstractBlocks*

List of delimited blocks.

BLOCK_TYPE
alias of *DelimitedBlock*

PREFIX = 'blockdef-'

load(*sections*)
Update blocks defined in 'sections' dictionary.

validate()

class leo.extensions.asciidoc.**Document**
Bases: object

backend

consume_attributes_and_comments(*comments_only=False*, *noblanks=False*)
Returns True if one or more attributes or comments were consumed. If 'noblanks' is True then consumation halts if a blank line is encountered.

doctype

getbackend()

getdoctype()

load_lang()
Load language configuration file.

parse_author(*s*)
Return False if the author is malformed.

parse_header(*doctype*, *backend*)
Parses header, sets corresponding document attributes and finalizes document doctype and backend properties. Returns False if the document does not have a header. 'doctype' and 'backend' are the doctype and backend option values passed on the command-line, None if no command-line option was not specified.

process_author_names()
Calculate any missing author related attributes.

set_deprecated_attribute(*old*, *new*)
Ensures the 'old' name of an attribute that was renamed to 'new' is still honored.

setbackend(*backend*)

setdoctype(*doctype*)

translate(*has_header*)

update_attributes(*attrs=None*)
Set implicit attributes and attributes in 'attrs'.

exception leo.extensions.asciidoc.**EAsciiDoc**
Bases: exceptions.Exception

class leo.extensions.asciidoc.**FloatingTitle**
Bases: *leo.extensions.asciidoc.Title*

Floated titles are translated differently.

static isnext()

static translate()

class leo.extensions.asciidoc.**Header**
Static methods and attributes only.

RCS_ID_RE = '^\\$Id: \\S+ (?P<revnumber>\\S+) (?P<revdate>\\S+) \\S+ (?P<author>\\S+)

REV_LINE_RE = '^(\\D*(?P<revnumber>.*?),)?(?P<revdate>.*?)(:\\s*(?P<revremark>.*))?$'

static parse()

class leo.extensions.asciidoc.**InsensitiveDict**
Bases: dict

Like a dictionary except key access is case insensitive. Keys are stored in lower case.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], **F) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class leo.extensions.asciidoc.**Lex**
Lexical analysis routines. Static methods and attributes only.

static canonical_subs()
Translate composite subs values.

static next()
Returns class of next element on the input (None if EOF). The reader is assumed to be at the first line following a previous element, end of file or line one. Exits with the reader pointing to the first line of the next element or EOF (leading blank lines are skipped).

prev_cursor = None

prev_element = None

static set_margin(*margin=0*)
Utility routine that sets the left margin to 'margin' space in a block of non-blank lines.

static subs(*options*)
Perform inline processing specified by 'options' (in 'options' order) on sequence of 'lines'.

static subs_1(*options*)
Perform substitution specified in 'options' (in 'options' order).

class leo.extensions.asciidoc.**List**
Bases: *leo.extensions.asciidoc.AbstractBlock*

NUMBER_STYLES = ('arabic', 'loweralpha', 'upperalpha', 'lowerroman', 'upperroman')

static calc_index(*style*)
Return the ordinal number of (1...) of the list item index for the given list style.

static calc_style()
Return the numbered list style ('arabic'...) of the list item index. Return None if unrecognized style.

check_index()
Check calculated self.ordinal (1,2,...) against the item number in the document (self.index) and check the number style is the same as the first item (self.number_style).

check_tags()
Check that all necessary tags are present.

dump()

isnext()

load(*name*, *entries*)

translate()

translate_entry()

translate_item()

validate()

class leo.extensions.asciidoc.**Lists**
Bases: *leo.extensions.asciidoc.AbstractBlocks*

List of List objects.

BLOCK_TYPE
alias of *List*

PREFIX = 'listdef-'

TAGS = ('list', 'entry', 'item', 'text', 'label', 'term')

TYPES = ('bulleted', 'numbered', 'labeled', 'callout')

dump()

initialize()

load(*sections*)

load_tags(*sections*)
Load listtags-* conf file sections to self.tags.

validate()

class leo.extensions.asciidoc.**Macro**

has_passthrough()

load(*entry*)

section_name(*name=None*)
Return macro markup template section name based on macro name and prefix. Return None section not found.

subs(*text*)

subs_passthroughs(*text*, *passthroughs*)
Replace macro attribute lists in text with placeholders. Substitute and append the passthrough attribute lists to the passthroughs list.

translate()
Block macro translation.

class leo.extensions.asciidoc.**Macros**

SYS_RE = '(?u)^(?P<name>[\\\\]?\\w(\\w|-)*?)::(?P<target>\\S*?)(\\[(?P<attrlist>.*?)

dump()

extract_passthroughs(*text*, *prefix=''*)
Extract the passthrough text and replace with temporary placeholders.

isnext()
Return matching macro if block macro is next on reader.

load(*entries*)

match(*prefix*, *name*, *text*)
Return re match object matching 'text' with macro type 'prefix', macro name 'name'.

restore_passthroughs(*text*)
Replace passthough placeholders with the original passthrough text.

subs(*text*, *prefix=''*, *callouts=False*)

validate()

class leo.extensions.asciidoc.**Message**
Message functions.

PROG = 'asciidoc'

deprecated(*msg*, *linenos=True*)

error(*msg*, *cursor=None*, *halt=False*)
Report fatal error. If halt=True raise EAsciiDoc exception. If halt=False don't exit application, continue in the hope of reporting all fatal errors finishing with a non-zero exit code.

format(*msg*, *prefix=''*, *linenos=True*, *cursor=None*, *offset=0*)
Return formatted message string.

stderr(*msg=''*)

stdout(*msg*)

unsafe(*msg*)

verbose(*msg*, *linenos=True*)

warning(*msg*, *linenos=True*, *offset=0*)

class leo.extensions.asciidoc.**OrderedDict**(*d=None*, ***kwargs*)
Bases: dict

Dictionary ordered by insertion order. Python Cookbook: Ordered Dictionary, Submitter: David Benjamin. http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/107747

clear() → None. Remove all items from D.

copy() → a shallow copy of D

items() → list of D's (key, value) pairs, as 2-tuples

keys() → list of D's keys

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → list of D's values

class leo.extensions.asciidoc.**Paragraph**
Bases: *leo.extensions.asciidoc.AbstractBlock*

dump()

isnext()

load(*name*, *entries*)

translate()

class leo.extensions.asciidoc.**Paragraphs**
Bases: *leo.extensions.asciidoc.AbstractBlocks*

List of paragraph definitions.

BLOCK_TYPE
alias of *Paragraph*

PREFIX = 'paradef-'

initialize()

load(*sections*)

validate()

class leo.extensions.asciidoc.**Plugin**
–filter and –theme option commands.

CMDS = ('install', 'remove', 'list', 'build')

static build()
Create plugin Zip file. args[0] is Zip file name. args[1] is plugin directory.

static get_dir()
Return plugins path (.asciidoc/filters or .asciidoc/themes) in user's home direcory or None if user home not defined.

static install()
Install plugin Zip file. args[0] is plugin zip file path. args[1] is optional destination plugins directory.

static list()
List all plugin directories (global and local).

static remove()
Delete plugin directory. args[0] is plugin name. args[1] is optional plugin directory (defaults to ~/.asciidoc/<plugin_name>).

type = None

class leo.extensions.asciidoc.**Reader**
Bases: *leo.extensions.asciidoc.Reader1*

Wraps (well, sought of) Reader1 class and implements conditional text inclusion.

eof()

read()

read_ahead(*count=1*)
Same as read_lines() but does not advance the file pointer.

read_lines(*count=1*)
Return tuple containing count lines.

read_next()

read_super()

read_until(*terminators*, *same_file=False*)
Like read() but reads lines up to (but not including) the first line that matches the terminator regular expression, regular expression object or list of regular expression objects. If same_file is True then the terminating pattern must occur in the file the was being read when the routine was called.

skip_blank_lines()

class leo.extensions.asciidoc.**Reader1**
Line oriented AsciiDoc input file reader. Processes include and conditional inclusion system macros. Tabs are expanded and lines are right trimmed.

READ_BUFFER_MIN = 10

close()

closefile()
Used by class methods to close nested include files.

eof()
Returns True if all lines have been read.

open(*fname*)

read(*skip=False*)
Read next line. Return None if EOF. Expand tabs. Strip trailing white space. Maintain self.next read ahead buffer. If skip=True then conditional exclusion is active (ifdef and ifndef macros).

read_next()
Like read() but does not advance file pointer.

unread(*cursor*)
Push the line (filename,linenumber,linetext) tuple back into the read buffer. Note that it's up to the caller to restore the previous cursor.

class leo.extensions.asciidoc.**Section**
Static methods and attributes only.

endtags = []

static gen_id()
The normalized value of the id attribute is an NCName according to the 'Namespaces in XML' Recommendation: NCName ::= NCNameStartChar NCNameChar* NCNameChar ::= NameChar - ':' NCNameStartChar ::= Letter | '_' NameChar ::= Letter | Digit | '.' | '-' | '_' | ':'

ids = []

static savetag(*etag*)
Save section end.

static set_id()

static setlevel()
Set document level and write open section close tags up to level.

static translate()

static translate_body()

class leo.extensions.asciidoc.**Table**
Bases: *leo.extensions.asciidoc.AbstractBlock*

ALIGN = {'<': 'left', '>': 'right', '^': 'center'}

FORMATS = ('psv', 'csv', 'dsv')

SEPARATORS = {'csv': ',', 'dsv': ':|\\n', 'psv': '((?<!\\S)((?P<span>[\\d.]+)(?P<o

VALIGN = {'<': 'top', '>': 'bottom', '^': 'middle'}

build_colspecs()
Generate column related substitution attributes.

dump()

get_style(*prefix*)
Return the style dictionary whose name starts with 'prefix'.

get_tags(*params*)

load(*name*, *entries*)

static parse_align_spec()
Parse AsciiDoc cell alignment specifier and return 2-tuple with horizonatal and vertical alignment names. Unspecified alignments set to None.

parse_cols(*cols*, *halign*, *valign*)
Build list of column objects from table 'cols', 'halign' and 'valign' attributes.

parse_csv(*text*)
Parse the table source text and return a list of rows, each row is a list of Cells.

parse_psv_dsv(*text*)
Parse list of PSV or DSV table source text lines and return a list of Cells.

parse_rows(*text*)
Parse the table source text into self.rows (a list of rows, each row is a list of Cells.

static parse_span_spec()
Parse AsciiDoc cell span specifier and return 2-tuple with horizonatal and vertical span counts. Set default values (1,1) if not specified.

subs_row(*row*, *rowtype*)
Substitute the list of Cells using the data tag. Returns a list of marked up table cell elements.

subs_rows(*rows*, *rowtype='body'*)
Return a string of output markup from a list of rows, each row is a list of raw data text.

translate()

validate()

validate_attributes()
Validate and parse table attributes.

class leo.extensions.asciidoc.**Table_OLD**
Bases: *leo.extensions.asciidoc.AbstractBlock*

ALIGNMENTS = {"'": 'right', '.': 'center', '`': 'left'}

COL_STOP = "(`|'|\\.)"

FORMATS = ('fixed', 'csv', 'dsv')

build_colspecs()
Generate colwidths and colspecs. This can only be done after the table arguments have been parsed since we use the table format.

dump()

isnext()

load(*name*, *entries*)

parse_csv(*rows*)
Parse the list of source table rows. Each row item in the returned list contains a list of cell data elements.

parse_dsv(*rows*)
Parse the list of source table rows. Each row item in the returned list contains a list of cell data elements.

parse_fixed(*rows*)
Parse the list of source table rows. Each row item in the returned list contains a list of cell data elements.

parse_rows(*rows*, *rtag*, *dtag*)
Parse rows list using the row and data tags. Returns a substituted list of output lines.

parse_ruler(*ruler*)
Parse ruler calculating underline and ruler column widths.

split_rows(*rows*)
Return a two item tuple containing a list of lines up to but not including the next underline (continued lines are joined) and the tuple of all lines after the underline.

subs_row(*data*, *dtag*)
Substitute the list of source row data elements using the data tag. Returns a substituted list of output table data items.

translate()

validate()

class leo.extensions.asciidoc.**Tables**
Bases: *leo.extensions.asciidoc.AbstractBlocks*

List of tables.

BLOCK_TYPE
alias of *Table*

PREFIX = 'tabledef-'

TAGS = ('colspec', 'headrow', 'footrow', 'bodyrow', 'headdata', 'footdata', 'bodydata'

dump()

load(*sections*)

load_tags(*sections*)
Load tabletags-* conf file sections to self.tags.

validate()

class leo.extensions.asciidoc.**Tables_OLD**
Bases: *leo.extensions.asciidoc.AbstractBlocks*

List of tables.

BLOCK_TYPE
alias of *Table_OLD*

PREFIX = 'old_tabledef-'

load(*sections*)

validate()

class leo.extensions.asciidoc.**Title**
Processes Header and Section titles. Static methods and attributes only.

attributes = {}

static dosubs()
Perform title substitutions.

static dump()

dump_dict = {}

static getnumber()
Return next section number at section 'level' formatted like 1.2.3.4.

static isnext()

level = 0

linecount = None

static load()
Load and validate [titles] section entries dictionary.

static parse()
Parse title at start of lines tuple.

pattern = None

section_numbers = [0, 0, 0, 0, 0]

sectname = None

static setsectname()
Set Title section name: If the first positional or 'template' attribute is set use it, next search for section title in [specialsections], if not found use default 'sect<level>' name.

subs = ()

static translate()
Parse the Title.attributes and Title.level from the reader. The real work has already been done by parse().

underlines = ('==', '--', '~~', '^^', '++')

class leo.extensions.asciidoc.**Trace**
Bases: object

Used in conjunction with the 'trace' attribute to generate diagnostic output. There is a single global instance of this class named trace.

SUBS_NAMES = ('specialcharacters', 'quotes', 'specialwords', 'replacements', 'attribut

class leo.extensions.asciidoc.**Writer**
Writes lines to output file.

close()

open(*fname*, *bom=None*)
bom is optional byte order mark. http://en.wikipedia.org/wiki/Byte-order_mark

write(**args*, ***kwargs*)
Iterates arguments, writes tuple and list arguments one line per element, else writes argument as single line. If no arguments writes blank line. If argument is None nothing is written. self.newline is appended to each line.

write_line(*line=None*)

write_tag(*tag*, *content*, *subs=None*, *d=None*, ***kwargs*)
Write content enveloped by tag. Substitutions specified in the 'subs' list are perform on the 'content'.

leo.extensions.asciidoc.**asciidoc**(*backend*, *doctype*, *confiles*, *infile*, *outfile*, *options*)
Convert AsciiDoc document to DocBook document of type doctype The AsciiDoc document is read from file object src the translated DocBook file written to file object dst.

leo.extensions.asciidoc.**assign**(*dst*, *src*)
Assign all attributes from 'src' object to 'dst' object.

leo.extensions.asciidoc.**char_decode**(*s*)

leo.extensions.asciidoc.**char_encode**(*s*)

leo.extensions.asciidoc.**char_encoding**()

leo.extensions.asciidoc.**char_len**(*s*)

leo.extensions.asciidoc.**column_width**(*s*)

leo.extensions.asciidoc.**create_zip**(*zip_file*, *src*, *skip_hidden=False*)
Create Zip file. If src is a directory archive all contained files and subdirectories, if src is a file archive the src file. Files and directories names starting with . are skipped if skip_hidden is True. Throws exception if error occurs.

leo.extensions.asciidoc.**date_str**(*t*)
Convert seconds since the Epoch to formatted local date string.

leo.extensions.asciidoc.**die**(*msg*)

leo.extensions.asciidoc.**dovetail**(*lines1*, *lines2*)
Append list or tuple of strings 'lines2' to list 'lines1'. Join the last non-blank item in 'lines1' with the first non-blank item in 'lines2' into a single string.

leo.extensions.asciidoc.**dovetail_tags**(*stag*, *content*, *etag*)
Merge the end tag with the first content line and the last content line with the end tag. This ensures verbatim elements don't include extraneous opening and closing line breaks.

leo.extensions.asciidoc.**dump_section**(*name*, *dict*, *f=<open file '<stdout>', mode 'w'>*)
Write parameters in 'dict' as in configuration file section format with section 'name'.

leo.extensions.asciidoc.**east_asian_widths = {'A': 1, 'F': 2, 'H': 1, 'N': 1, 'Na': 1, 'W'**
Mapping of result codes from *unicodedata.east_asian_width()* to character column widths.

leo.extensions.asciidoc.**execute**(*cmd*, *opts*, *args*)
Execute asciidoc with command-line options and arguments. cmd is asciidoc command or asciidoc.py path. opts and args conform to values returned by getopt.getopt(). Raises SystemExit if an error occurs.

Doctests:

1. Check execution:

```
>>> import StringIO
>>> infile = StringIO.StringIO('Hello *{author}*')
>>> outfile = StringIO.StringIO()
>>> opts = []
>>> opts.append(('--backend','html4'))
>>> opts.append(('--no-header-footer',None))
>>> opts.append(('--attribute','author=Joe Bloggs'))
>>> opts.append(('--out-file',outfile))
>>> execute(__file__, opts, [infile])
>>> print outfile.getvalue()
<p>Hello <strong>Joe Bloggs</strong></p>
```

```
>>>
```

leo.extensions.asciidoc.**extract_zip**(*zip_file*, *destdir*)
Unzip Zip file to destination directory. Throws exception if error occurs.

leo.extensions.asciidoc.**file_in**(*fname*, *directory*)
Return True if file fname resides inside directory.

leo.extensions.asciidoc.**filter_lines**(*filter_cmd*, *lines*, *attrs={}*)
Run 'lines' through the 'filter_cmd' shell command and return the result. The 'attrs' dictionary contains additional filter attributes.

leo.extensions.asciidoc.**get_args**(*val*)

leo.extensions.asciidoc.**get_kwargs**(*val*)

leo.extensions.asciidoc.**is_array**(*obj*)
Return True if object is list or tuple type.

leo.extensions.asciidoc.**is_attr_defined**(*attrs*, *dic*)
Check if the sequence of attributes is defined in dictionary 'dic'. Valid 'attrs' sequence syntax: <attr> Return True if single attrbiute is defined. <attr1>,<attr2>,... Return True if one or more attributes are defined. <attr1>+<attr2>+... Return True if all the attributes are defined.

leo.extensions.asciidoc.**is_name**(*s*)
Return True if s is valid attribute, macro or tag name (starts with alpha containing alphanumeric and dashes only).

leo.extensions.asciidoc.**is_re**(*s*)
Return True if s is a valid regular expression else return False.

leo.extensions.asciidoc.**is_safe_file**(*fname*, *directory=None*)

leo.extensions.asciidoc.**join_lines_OLD**(*lines*)
Return a list in which lines terminated with the backslash line continuation character are joined.

leo.extensions.asciidoc.**localapp**()
Return True if we are not executing the system wide version i.e. the configuration is in the executable's directory.

leo.extensions.asciidoc.**lstrip_list**(*s*)
Return list with empty items from start of list removed.

leo.extensions.asciidoc.**parse_attributes**(*attrs*, *dict*)
Update a dictionary with name/value attributes from the attrs string. The attrs string is a comma separated list of values and keyword name=value pairs. Values must preceed keywords and are named '1','2'... The entire attributes list is named '0'. If keywords are specified string values must be quoted. Examples:

attrs: '' dict: {}

attrs: 'hello,world' dict: {'2': 'world', '0': 'hello,world', '1': 'hello'}

attrs: '"hello", planet="earth"' dict: {'planet': 'earth', '0': '"hello",planet="earth"', '1': 'hello'}

leo.extensions.asciidoc.**parse_entries**(*entries*, *dict*, *unquote=False*, *unique_values=False*, *allow_name_only=False*, *escape_delimiter=True*)
Parse name=value entries from from lines of text in 'entries' into dictionary 'dict'. Blank lines are skipped.

leo.extensions.asciidoc.**parse_entry**(*entry*, *dict=None*, *unquote=False*, *unique_values=False*, *allow_name_only=False*, *escape_delimiter=True*)
Parse name=value entry to dictionary 'dict'. Return tuple (name,value) or None if illegal entry. If name= then value is set to ''. If name and allow_name_only=True then value is set to ''. If name! and allow_name_only=True then value is set to None. Leading and trailing white space is striped from 'name' and 'value'. 'name' can contain any printable characters. If the '=' delimiter character is allowed in the 'name' then it must be escaped with a backslash and escape_delimiter must be True. If 'unquote' is True leading and trailing double-quotes are stripped from 'name' and 'value'. If unique_values' is True then dictionary entries with the same value are removed before the parsed entry is added.

leo.extensions.asciidoc.**parse_list**(*s*)
Parse comma separated string of Python literals. Return a tuple of of parsed values.

leo.extensions.asciidoc.**parse_named_attributes**(*s*, *attrs*)
Update a attrs dictionary with name="value" attributes from the s string. Returns False if invalid syntax. Example: attrs: 'star="sun",planet="earth"' dict: {'planet':'earth', 'star':'sun'}

leo.extensions.asciidoc.**parse_options**(*options*, *allowed*, *errmsg*)
Parse comma separated string of unquoted option names and return as a tuple of valid options. 'allowed' is a list of allowed option values. If allowed=() then all legitimate names are allowed. 'errmsg' is an error message prefix if an illegal option error is thrown.

leo.extensions.asciidoc.**parse_to_list**(*val*)

leo.extensions.asciidoc.**re_join**(*relist*)
Join list of regular expressions re1,re2,... to single regular expression (re1)|(re2)|...

leo.extensions.asciidoc.**rstrip_list**(*s*)
Return list with empty items from end of list removed.

leo.extensions.asciidoc.**safe**()

leo.extensions.asciidoc.**safe_filename**(*fname*, *parentdir*)
Return file name which must reside in the parent file directory. Return None if file is not safe.

leo.extensions.asciidoc.**show_help**(*topic*, *f=None*)
Print help topic to file object f.

leo.extensions.asciidoc.**strip_list**(*s*)
Return list with empty items from start and end of list removed.

leo.extensions.asciidoc.**strip_quotes**(*s*)
Trim white space and, if necessary, quote characters from s.

leo.extensions.asciidoc.**subs_attrs**(*lines*, *dictionary=None*)
Substitute 'lines' of text with attributes from the global document.attributes dictionary and from 'dictionary' ('dictionary' entries take precedence). Return a tuple of the substituted lines. 'lines' containing undefined attributes are deleted. If 'lines' is a string then return a string.

- Attribute references are substituted in the following order: simple, conditional, system.
- Attribute references inside 'dictionary' entry values are substituted.

leo.extensions.asciidoc.**subs_quotes**(*text*)
Quoted text is marked up and the resulting text is returned.

leo.extensions.asciidoc.**subs_tag**(*tag*, *dict={}*)
Perform attribute substitution and split tag string returning start, end tag tuple (c.f. Config.tag()).

leo.extensions.asciidoc.**symbolize**(*s*)
Drop non-symbol characters and convert to lowercase.

leo.extensions.asciidoc.**system**(*name*, *args*, *is_macro=False*, *attrs=None*)
Evaluate a system attribute ({name:args}) or system block macro (name::[args]). If is_macro is True then we are processing a system block macro otherwise it's a system attribute. The attrs dictionary is updated by the counter and set system attributes. NOTE: The include1 attribute is used internally by the include1::[] macro and is not for public use.

leo.extensions.asciidoc.**time_str**(*t*)
Convert seconds since the Epoch to formatted local time string.

leo.extensions.asciidoc.**update_attrs**(*attrs*, *dict*)
Update 'attrs' dictionary with parsed attributes in dictionary 'dict'.

leo.extensions.asciidoc.**usage**(*msg=''*)

leo.extensions.asciidoc.**userdir**()
Return user's home directory or None if it is not defined.

1.2.2.3 colors Module

Colors module provides a colors names database and functions to access it.

The database is a python dictionary with reduced colornames as keys and color data in the form '#RRGGBB' as values.

The reudced color names are common color names with spaces and capitalization removed.

Functions to access the data base are:

This function will convert *name* and return it in '#RRGGBB' format if possible otherwise it will do the sme for *default*.

If default can not be converted, None will be returned.

name and *default* are reduced by removing spaces and capitalization before looking them up in the database.

This function will first call getColor to convert *name* (or *default*) to '#RRGGBB' format, then it will convert this and return it as a python tuple in the form (0-255, 0-255, 0-255).

Returns None if anything goes wrong.

leo.extensions.colors.**getColor**(*name*, *default=None*)

leo.extensions.colors.**getColorRGB**(*name*, *default=None*)

1.2.2.4 patch_11_01 Module

Patch utility to apply unified diffs

Brute-force line-by-line non-recursive parsing

Copyright (c) 2008-2011 anatoly techtonik Available under the terms of MIT license

Project home: http://code.google.com/p/python-patch/

$Id: patch.py 117 2011-01-09 16:38:03Z techtonik $ $HeadURL: https://python-patch.googlecode.com/svn/trunk/patch.py $

class leo.extensions.patch_11_01.**Hunk**
Bases: object

Parsed hunk data container (hunk starts with @@ -R +R @@)

copy()

startsrc = None
line count starts with 1

class leo.extensions.patch_11_01.**Patch**(*stream=None*)
Bases: object

apply()
apply parsed patch return True on success

can_patch(*filename*)
Check if specified filename can be patched. Returns None if file can not be found among source filenames. False if patch can not be applied clearly. True otherwise.

Returns True, False or None

copy()

header = None
headers for each file

hunkends = None
file endings statistics for every hunk

hunks = None
list of lists of hunks

parse(*stream*)
parse unified diff

patch_stream(*instream*, *hunks*)
Generator that yields stream patched with hunks iterable

Converts lineends in hunk lines to the best suitable format autodetected from input

source = None
list of source filenames

type = None
patch type - one of constants

write_hunks(*srcname*, *tgtname*, *hunks*)

leo.extensions.patch_11_01.**fromfile**(*filename*)
Parse patch file and return Patch() object

leo.extensions.patch_11_01.**fromstring**(*s*)
Parse text string and return Patch() object

leo.extensions.patch_11_01.**fromurl**(*url*)
Read patch from URL

1.2.2.5 sh Module

1.2.2.6 testExtension Module

1.2.3 external Package

1.2.3.1 external Package

1.2.3.2 codewise Module

CodeWise - global code intelligence database

Why this module

- Exuberant ctags is an excellent code scanner
- Unfortunately, TAGS file lookup sucks for "find methods of this class"
- TAGS files can be all around the hard drive. CodeWise database is just one file (by default ~/.codewise.db)
- I wanted to implement modern code completion for Leo editor
- codewise.py is usable as a python module, or a command line tool.

Creating ctags data

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. **[Optional] Create a custom ~/.ctags file containing default** configuration settings for ctags. See: http://ctags.sourceforge.net/ctags.html#FILES for more details.

   The `codewise setup` command (see below), will leave this file alone if it exists; otherwise, `codewise setup` will create a ~/.ctags file containing:

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

3. Create the ctags data in ~/.codewise.db using this module. Execute the following from a console window:

   ```
   codewise setup
       # Optional: creates ~/.ctags if it does not exist.
       # See http://ctags.sourceforge.net/ctags.html#FILES
   codewise init
       # Optional: deletes ~/.codewise.db if it exists.
   codewise parse <path to directory>
       # Adds ctags data to ~/.codewise.db for <directory>
   ```

Note: On Windows, use a batch file, say codewise.bat, to execute the above code. codewise.bat contains:

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

Using the autocompleter

After restarting Leo, type, for example, in the body pane:

```
c.op<ctrl-space>
```

that is, use use the autocomplete-force command, to find all the c. methods starting with 'op' etc.

Theory of operation

- ~/.codewise.db is an sqlite database with following tables:

CLASS maps class id's to names.

FILE maps file id's to file names

DATASOURCE contains places where data has been parsed from, to enable reparse

FUNCTION, the most important one, contains functions/methods, along with CLASS and FILE it was found in. Additionally, it has SEARCHPATTERN field that can be used to give calltips, or used as a regexp to find the method from file quickly.

You can browse the data by installing sqlitebrovser and doing 'sqlitebrowser ~/codewise.db'

If you know the class name you want to find the methods for, CodeWise.get_members with a list of classes to match.

If you want to match a function without a class, call CodeWise.get_functions. This can be much slower if you have a huge database.

class leo.external.codewise.**CodeWise**(*dbpath=None*)
Bases: object

add_source(*type*, *src*)

class_id(*classname*)
return class id. May create new class

create_caches()
read existing db and create caches

createdb(*dbpath*)

cursor()

feed_ctags(*tagsfile_obj*)

feed_function(*func_name*, *class_name*, *file_name*, *aux*)
insert one function

'aux' can be a search pattern (as with ctags), signature, or description

feed_scintilla(*apifile_obj*)
handle scintilla api files

Syntax is like:

qt.QApplication.style?4() -> QStyle

file_id(*fname*)

get_functions(*prefix=None*)

get_members(*classnames*)

parse(*paths*)

parseall()

reset_caches()

sources()

zap_symbols()

class leo.external.codewise.**ContextSniffer**
Bases: object

Class to analyze surrounding context and guess class

For simple dynamic code completion engines

declare(*var*, *klass*)

push_declarations(*body*)

set_small_context(*body*)
Set immediate function

leo.external.codewise.**callers**(*n=4*, *count=0*, *excludeCaller=True*, *files=False*)
Return a list containing the callers of the function that called callerList.

If the excludeCaller keyword is True (the default), callers is not on the list.

If the files keyword argument is True, filenames are included in the list.

leo.external.codewise.**cmd_functions**(*args*)

leo.external.codewise.**cmd_init**(*args*)

leo.external.codewise.**cmd_members**(*args*)

leo.external.codewise.**cmd_parse**(*args*)

leo.external.codewise.**cmd_parseall**(*args*)

leo.external.codewise.**cmd_scintilla**(*args*)

leo.external.codewise.**cmd_setup**(*args*)

leo.external.codewise.**cmd_tags**(*args*)

leo.external.codewise.**doKeywordArgs**(*keys*, *d=None*)
Return a result dict that is a copy of the keys dict with missing items replaced by defaults in d dict.

leo.external.codewise.**error**(**args*, ***keys*)

leo.external.codewise.**es_exception**(*full=True*, *c=None*, *color='red'*)

leo.external.codewise.**getLastTracebackFileAndLineNumber**()

leo.external.codewise.**isBytes**(*s*)
Return True if s is Python3k bytes type.

leo.external.codewise.**isCallable**(*obj*)

leo.external.codewise.**isString**(*s*)
Return True if s is any string, but not bytes.

leo.external.codewise.**isUnicode**(*s*)
Return True if s is a unicode string.

leo.external.codewise.**isValidEncoding**(*encoding*)

leo.external.codewise.**main**()

leo.external.codewise.**pdb**(*message=''*)
Fall into pdb.

leo.external.codewise.**pr**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane. The first, third, fifth, etc. arg translated by translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.external.codewise.**printlines**(*lines*)

leo.external.codewise.**run_ctags**(*paths*)

leo.external.codewise.**shortFileName**(*fileName*, *n=None*)
Return the base name of a path.

leo.external.codewise.**test**(*self*)

leo.external.codewise.**toEncodedString**(*s*, *encoding='utf-8'*, *reportErrors=False*)
Convert unicode string to an encoded string.

leo.external.codewise.**toUnicode**(*s*, *encoding='utf-8'*, *reportErrors=False*)
Connvert a non-unicode string with the given encoding to unicode.

leo.external.codewise.**trace**(**args*, ***keys*)

leo.external.codewise.**translateArgs**(*args*, *d*)
Return the concatenation of all args, with odd args translated.

leo.external.codewise.**u**(*s*)

leo.external.codewise.**ue**(*s*, *encoding*)

1.2.3.3 edb Module

edb: The Python Debugger Pdb, modified for blender by EKR

To use the debugger in its simplest form:

```
>>> import pdb
>>> pdb.run('<a statement>')
```

The debugger's prompt is '(Pdb) '. This will stop in the first function call in <a statement>.

Alternatively, if a statement terminated with an unhandled exception, you can use pdb's post-mortem facility to inspect the contents of the traceback:

```
>>> <a statement>
<exception traceback>
>>> import pdb
>>> pdb.pm()
```

The commands recognized by the debugger are listed in the next section. Most can be abbreviated as indicated; e.g., h(elp) means that 'help' can be typed as 'h' or 'help' (but not as 'he' or 'hel', nor as 'H' or 'Help' or 'HELP'). Optional arguments are enclosed in square brackets. Alternatives in the command syntax are separated by a vertical bar (|).

A blank line repeats the previous command literally, except for 'list', where it lists the next 11 lines.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point ('!'). This is a powerful way to inspect the program being debugged; it is even possible to change variables or call functions. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

The debugger supports aliases, which can save typing. And aliases can have parameters (see the alias help entry) which allows one a certain level of adaptability to the context under examination.

Multiple commands may be entered on a single line, separated by the pair ';;'. No intelligence is applied to separating the commands; the input is split at the first ';;', even if it is in the middle of a quoted string.

If a file ".pdbrc" exists in your home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overriden by the local file.

Aside from aliases, the debugger is not directly programmable; but it is implemented as a class from which you can derive your own debugger class, which you can make as fancy as you like.

Debugger commands

h(elp) Without argument, print the list of available commands. With a command name as argument, print help about that command. "help pdb" shows the full pdb documentation. "help exec" gives help on the ! command.

w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the "current frame", which determines the context of most commands. 'bt' is an alias for this command.

d(own) [count] Move the current frame count (default one) levels down in the stack trace (to a newer frame).

u(p) [count] Move the current frame count (default one) levels up in the stack trace (to an older frame).

b(reak) [([filename:]lineno | function) [, condition]] Without argument, list all breaks.

With a line number argument, set a break at this line in the current file. With a function name, set a break at the first executable line of that function. If a second argument is present, it is a string specifying an expression which must evaluate to true before the breakpoint is honored.

The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched for on sys.path; the .py suffix may be omitted.

tbreak [([filename:]lineno | function) [, condition]] Same arguments as break, but sets a temporary breakpoint: it is automatically deleted when first hit.

cl(ear) filename:lineno cl(ear) [bpnumber [bpnumber. . .]]

With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation). With a filename:lineno argument, clear all breaks at that line in that file.

disable bpnumber [bpnumber . . .] Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable bpnumber [bpnumber . . .] Enables the breakpoints given as a space separated list of breakpoint numbers.

ignore bpnumber [count] Set the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition] Set a new condition for the breakpoint, an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

commands [bpnumber] (com) . . . (com) end (Pdb)

Specify a list of commands for breakpoint number bpnumber. The commands themselves are entered on the following lines. Type a line containing just 'end' to terminate the commands. The commands are executed when the breakpoint is hit.

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no bpnumber argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint – which could have its own command list, leading to ambiguities about which list to execute.

If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you will see no sign that the breakpoint was reached.

s(tep) Execute the current line, stop at the first possible occasion (either in a function that is called or in the current function).

n(ext) Continue execution until the next line in the current function is reached or it returns.

unt(il) [lineno] Without argument, continue execution until the line with a number greater than the current one is reached. With a line number, continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

j(ump) lineno Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a for loop or out of a finally clause.

r(eturn) Continue execution until the current function returns.

retval Print the return value for the last return of a function.

run [args...] Restart the debugged python program. If a string is supplied it is splitted with "shlex", and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. "restart" is an alias for "run".

c(ont(inue)) Continue execution, only stop when a breakpoint is encountered.

l(ist) [first [,last] | .]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With . as argument, list 11 lines around the current line. With one argument, list 11 lines starting at that line. With two arguments, list the given range; if the second argument is less than the first, it is a count.

The current line in the current frame is indicated by "->". If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by ">>", if it differs from the current line.

longlist | ll List the whole source code for the current function or frame.

a(rgs) Print the argument list of the current function.

p(rint) expression Print the value of the expression.

pp expression Pretty-print the value of the expression.

whatis arg Print the type of the argument.

source expression Try to get source code for the given object and display it.

display [expression]

Display the value of the expression if it changed, each time execution stops in the current frame.

Without expression, list all display expressions for the current frame.

undisplay [expression]

Do not display the expression any more in the current frame.

Without expression, clear all display expressions for the current frame.

interact

Start an interative interpreter whose global namespace contains all the (global and local) names found in the current scope.

alias [name [command [parameter parameter ...]]] Create an alias called 'name' that executes 'command'. The command must *not* be enclosed in quotes. Replaceable parameters can be indicated by %1, %2, and so on, while %* is replaced by all the parameters. If no command is given, the current alias for name is shown. If no name is given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note! You *can* override internal pdb commands with aliases! Those internal commands are then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the .pdbrc file):

Print instance variables (usage "pi classInst") alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k] # Print instance variables in self alias ps pi self

unalias name Delete the specified alias.

debug code Enter a recursive debugger that steps through the code argument (which is an arbitrary expression or statement to be executed in the current environment).

q(uit) exit

Quit from the debugger. The program being executed is aborted.

(!) statement Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To assign to a global variable you must always prefix the command with a 'global' command, e.g.: (Pdb) global list_options; list_options = ['-l'] (Pdb)

`leo.external.edb.`**`run`**(*statement*, *globals=None*, *locals=None*)

`leo.external.edb.`**`pm`**()

`class` `leo.external.edb.`**`Pdb`**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*)

Bases: `bdb.Bdb`, `cmd.Cmd`

`bp_commands`(*frame*)

Call every command that was set for the current active breakpoint (if there is one).

Returns True if the normal interaction function must be called, False otherwise.

`break_here`(*frame*)

`checkline`(*filename*, *lineno*)

Check whether specified line seems to be executable.

Return *lineno* if it is, 0 if not (e.g. a docstring, comment, blank line or EOF). Warning: testing is not comprehensive.

`commands_resuming = ['do_continue', 'do_step', 'do_next', 'do_return', 'do_quit', 'do_`

`default`(*line*)

`defaultFile`()

Produce a reasonable default.

`displayhook`(*obj*)

Custom displayhook for the exec in default(), which prevents assignment of the _ variable in the builtins.

`do_EOF`(*arg*)

EOF Handles the receipt of EOF as a command.

`do_a`(*arg*)

a(rgs) Print the argument list of the current function.

`do_alias`(*arg*)

alias [name [command [parameter parameter ...]]] Create an alias called 'name' that executes 'command'. The command must *not* be enclosed in quotes. Replaceable parameters can be indicated by %1, %2, and so on, while %* is replaced by all the parameters. If no command is given, the current alias for name is shown. If no name is given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note! You *can* override internal pdb commands with aliases! Those internal commands are then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the .pdbrc file):

Print instance variables (usage "pi classInst") alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k] # Print instance variables in self alias ps pi self

do_args(*arg*)
a(rgs) Print the argument list of the current function.

do_b(*arg*, *temporary=0*)
b(reak) [([filename:]lineno | function) [, condition]] Without argument, list all breaks.

With a line number argument, set a break at this line in the current file. With a function name, set a break at the first executable line of that function. If a second argument is present, it is a string specifying an expression which must evaluate to true before the breakpoint is honored.

The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched for on sys.path; the .py suffix may be omitted.

do_break(*arg*, *temporary=0*)
b(reak) [([filename:]lineno | function) [, condition]] Without argument, list all breaks.

With a line number argument, set a break at this line in the current file. With a function name, set a break at the first executable line of that function. If a second argument is present, it is a string specifying an expression which must evaluate to true before the breakpoint is honored.

The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched for on sys.path; the .py suffix may be omitted.

do_bt(*arg*)
w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the "current frame", which determines the context of most commands. 'bt' is an alias for this command.

do_c(*arg*)
c(ont(inue)) Continue execution, only stop when a breakpoint is encountered.

do_cl(*arg*)
cl(ear) filename:lineno cl(ear) [bpnumber [bpnumber...]]

With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation). With a filename:lineno argument, clear all breaks at that line in that file.

do_clear(*arg*)
cl(ear) filename:lineno cl(ear) [bpnumber [bpnumber...]]

With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation). With a filename:lineno argument, clear all breaks at that line in that file.

do_commands(*arg*)
commands [bpnumber] (com) ... (com) end (Pdb)

Specify a list of commands for breakpoint number bpnumber. The commands themselves are entered on the following lines. Type a line containing just 'end' to terminate the commands. The commands are executed when the breakpoint is hit.

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no bpnumber argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint – which could have its own command list, leading to ambiguities about which list to execute.

If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you will see no sign that the breakpoint was reached.

do_condition(*arg*)
condition bpnumber [condition] Set a new condition for the breakpoint, an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

do_cont(*arg*)
c(ont(inue)) Continue execution, only stop when a breakpoint is encountered.

do_continue(*arg*)
c(ont(inue)) Continue execution, only stop when a breakpoint is encountered.

do_d(*arg*)
d(own) [count] Move the current frame count (default one) levels down in the stack trace (to a newer frame).

do_debug(*arg*)
debug code Enter a recursive debugger that steps through the code argument (which is an arbitrary expression or statement to be executed in the current environment).

do_disable(*arg*)
disable bpnumber [bpnumber ...] Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

do_display(*arg*)
display [expression]

Display the value of the expression if it changed, each time execution stops in the current frame.

Without expression, list all display expressions for the current frame.

do_down(*arg*)
d(own) [count] Move the current frame count (default one) levels down in the stack trace (to a newer frame).

do_enable(*arg*)
enable bpnumber [bpnumber ...] Enables the breakpoints given as a space separated list of breakpoint numbers.

do_exit(*arg*)
q(uit) exit

Quit from the debugger. The program being executed is aborted.

do_h(*arg*)
h(elp) Without argument, print the list of available commands. With a command name as argument, print help about that command. "help pdb" shows the full pdb documentation. "help exec" gives help on the ! command.

do_help(*arg*)
h(elp) Without argument, print the list of available commands. With a command name as argument, print help about that command. “help pdb” shows the full pdb documentation. “help exec” gives help on the ! command.

do_ignore(*arg*)
ignore bpnumber [count] Set the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

do_interact(*arg*)
interact

Start an interative interpreter whose global namespace contains all the (global and local) names found in the current scope.

do_j(*arg*)
j(ump) lineno Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don’t want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a for loop or out of a finally clause.

do_jump(*arg*)
j(ump) lineno Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don’t want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a for loop or out of a finally clause.

do_l(*arg*)
l(ist) [first [,last] | .]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With . as argument, list 11 lines around the current line. With one argument, list 11 lines starting at that line. With two arguments, list the given range; if the second argument is less than the first, it is a count.

The current line in the current frame is indicated by “->”. If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by “>>”, if it differs from the current line.

do_list(*arg*)
l(ist) [first [,last] | .]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With . as argument, list 11 lines around the current line. With one argument, list 11 lines starting at that line. With two arguments, list the given range; if the second argument is less than the first, it is a count.

The current line in the current frame is indicated by “->”. If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by “>>”, if it differs from the current line.

do_ll(*arg*)
longlist | ll List the whole source code for the current function or frame.

do_longlist(*arg*)
longlist | ll List the whole source code for the current function or frame.

do_n(*arg*)
n(ext) Continue execution until the next line in the current function is reached or it returns.

do_next(*arg*)
n(ext) Continue execution until the next line in the current function is reached or it returns.

do_p(*arg*)
p(rint) expression Print the value of the expression.

do_pp(*arg*)
pp expression Pretty-print the value of the expression.

do_print(*arg*)
p(rint) expression Print the value of the expression.

do_q(*arg*)
q(uit) exit

Quit from the debugger. The program being executed is aborted.

do_quit(*arg*)
q(uit) exit

Quit from the debugger. The program being executed is aborted.

do_r(*arg*)
r(eturn) Continue execution until the current function returns.

do_restart(*arg*)
run [args...] Restart the debugged python program. If a string is supplied it is splitted with "shlex", and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. "restart" is an alias for "run".

do_return(*arg*)
r(eturn) Continue execution until the current function returns.

do_retval(*arg*)
retval Print the return value for the last return of a function.

do_run(*arg*)
run [args...] Restart the debugged python program. If a string is supplied it is splitted with "shlex", and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. "restart" is an alias for "run".

do_rv(*arg*)
retval Print the return value for the last return of a function.

do_s(*arg*)
s(tep) Execute the current line, stop at the first possible occasion (either in a function that is called or in the current function).

do_source(*arg*)
source expression Try to get source code for the given object and display it.

do_step(*arg*)
s(tep) Execute the current line, stop at the first possible occasion (either in a function that is called or in the current function).

do_tbreak(*arg*)
tbreak [([filename:]lineno | function) [, condition]] Same arguments as break, but sets a temporary breakpoint: it is automatically deleted when first hit.

do_u(*arg*)
u(p) [count] Move the current frame count (default one) levels up in the stack trace (to an older frame).

do_unalias(*arg*)
unalias name Delete the specified alias.

do_undisplay(*arg*)
undisplay [expression]

Do not display the expression any more in the current frame.

Without expression, clear all display expressions for the current frame.

do_unt(*arg*)
unt(il) [lineno] Without argument, continue execution until the line with a number greater than the current one is reached. With a line number, continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

do_until(*arg*)
unt(il) [lineno] Without argument, continue execution until the line with a number greater than the current one is reached. With a line number, continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

do_up(*arg*)
u(p) [count] Move the current frame count (default one) levels up in the stack trace (to an older frame).

do_w(*arg*)
w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the "current frame", which determines the context of most commands. 'bt' is an alias for this command.

do_whatis(*arg*)
whatis arg Print the type of the argument.

do_where(*arg*)
w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the "current frame", which determines the context of most commands. 'bt' is an alias for this command.

error(*msg*)

execRcLines()

forget()

format_stack_entry(*frame_lineno*, *lprefix=': '*)

handle_command_def(*line*)
Handles one command line during command list definition.

help_exec()
(!) statement Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To assign to a global variable you must always prefix the command with a 'global' command, e.g.: (Pdb) global list_options; list_options = ['-l'] (Pdb)

help_pdb()

interaction(*frame*, *traceback*)

lineinfo(*identifier*)

lookupmodule(*filename*)
Helper function for break/clear parsing – may be overridden.

lookupmodule() translates (possibly incomplete) file or module name into an absolute file name.

message(*msg*)

onecmd(*line*)
Interpret the argument as though it had been typed in response to the prompt.

Checks whether this line is typed at the normal prompt or in a breakpoint command list definition.

precmd(*line*)
Handle alias expansion and ';;' separator.

preloop()

print_stack_entry(*frame_lineno*, *prompt_prefix='\n-> '*)

print_stack_trace()

reset()

setup(*f*, *tb*)

sigint_handler(*signum*, *frame*)

user_call(*frame*, *argument_list*)
This method is called when there is the remote possibility that we ever need to stop in this function.

user_exception(*frame*, *exc_info*)
This function is called if an exception occurs, but only if we are to stop at or just below this level.

user_line(*frame*)
This function is called when we stop or break at this line.

user_return(*frame*, *return_value*)
This function is called when a return trap is set here.

leo.external.edb.**runeval**(*expression*, *globals=None*, *locals=None*)

leo.external.edb.**runctx**(*statement*, *globals*, *locals*)

leo.external.edb.**runcall**(**args*, ***kwds*)

leo.external.edb.**set_trace**()

leo.external.edb.**post_mortem**(*t=None*)

leo.external.edb.**help**()

1.2.3.4 ipy_leo Module

1.2.3.5 leoSAGlobals Module

leoSAGlobals.py: the stand-alone version of leo.core.leoGlobals.py

class leo.external.leoSAGlobals.**Bunch**(***keywords*)
Bases: object

A class that represents a colection of things.

Especially useful for representing a collection of related variables.

get(*key*, *theDefault=None*)

ivars()

keys()

toString()

leo.external.leoSAGlobals.**CheckVersion**(*s1*, *s2*, *condition='>='*, *stringCompare=None*, *delimiter='.'*, *trace=False*)

leo.external.leoSAGlobals.**CheckVersionToInt**(*s*)

leo.external.leoSAGlobals.**adjustTripleString**(*s*, *tab_width*)
Remove leading indentation from a triple-quoted string.

This works around the fact that Leo nodes can't represent underindented strings.

leo.external.leoSAGlobals.**angleBrackets**(*s*)

leo.external.leoSAGlobals.**appendToList**(*out*, *s*)

leo.external.leoSAGlobals.**bunch**
alias of *leo.external.leoSAGlobals.Bunch*

leo.external.leoSAGlobals.**callers**(*n=4*, *count=0*, *excludeCaller=True*, *files=False*)
Return a list containing the callers of the function that called callerList.

If the excludeCaller keyword is True (the default), callers is not on the list.

If the files keyword argument is True, filenames are included in the list.

leo.external.leoSAGlobals.**choose**(*cond*, *a*, *b*)

leo.external.leoSAGlobals.**cls**()
Clear the screen.

leo.external.leoSAGlobals.**computeLeadingWhitespace**(*width*, *tab_width*)

leo.external.leoSAGlobals.**computeWidth**(*s*, *tab_width*)

leo.external.leoSAGlobals.**computeWindowTitle**(*fileName*)

leo.external.leoSAGlobals.**convertPythonIndexToRowCol**(*s*, *i*)
Convert index i into string s into zero-based row/col indices.

leo.external.leoSAGlobals.**convertRowColToPythonIndex**(*s*, *row*, *col*, *lines=None*)
Convert zero-based row/col indices into a python index into string s.

leo.external.leoSAGlobals.**dictToString**(*d*, *tag=None*, *verbose=True*, *indent=''*)

leo.external.leoSAGlobals.**doKeywordArgs**(*keys*, *d=None*)
Return a result dict that is a copy of the keys dict with missing items replaced by defaults in d dict.

leo.external.leoSAGlobals.**ensureLeadingNewlines**(*s*, *n*)

leo.external.leoSAGlobals.**ensureTrailingNewlines**(*s*, *n*)

leo.external.leoSAGlobals.**error**(**args*, ***keys*)

leo.external.leoSAGlobals.**es**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane. The first, third, fifth, etc. arg translated by translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.external.leoSAGlobals.**es_error**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane. The first, third, fifth, etc. arg translated by translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.external.leoSAGlobals.**es_exception**(*full=True*, *c=None*, *color='red'*)

leo.external.leoSAGlobals.**es_exception_type**(*c=None*, *color='red'*)

leo.external.leoSAGlobals.**es_print**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane. The first, third, fifth, etc. arg translated by translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.external.leoSAGlobals.**es_print_error**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane. The first, third, fifth, etc. arg translated by translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.external.leoSAGlobals.**es_print_exception**(*full=True*, *c=None*, *color='red'*)

leo.external.leoSAGlobals.**es_trace**(**args*, ***keys*)

leo.external.leoSAGlobals.**escaped**(*s*, *i*)

class leo.external.leoSAGlobals.**fileLikeObject**(*encoding='utf-8'*, *fromString=None*)
Define a file-like object for redirecting writes to a string.

The caller is responsible for handling newlines correctly.

clear()

close()

flush()

get()

getvalue()

read()

readline()

write(*s*)

leo.external.leoSAGlobals.**find_line_start**(*s*, *i*)

leo.external.leoSAGlobals.**find_on_line**(*s*, *i*, *pattern*)

leo.external.leoSAGlobals.**flattenList**(*theList*)

leo.external.leoSAGlobals.**funcToMethod**(*f*, *theClass*, *name=None*)

leo.external.leoSAGlobals.**getDocString**(*s*)
Return the text of the first docstring found in s.

leo.external.leoSAGlobals.**getDocStringForFunction**(*func*)
Return the docstring for a function that creates a Leo command.

leo.external.leoSAGlobals.**getLastTracebackFileAndLineNumber**()

leo.external.leoSAGlobals.**getLine**(*s*, *i*)
Return i,j such that s[i:j] is the line surrounding s[i]. s[i] is a newline only if the line is empty. s[j] is a newline unless there is no trailing newline.

leo.external.leoSAGlobals.**getLineAfter**(*s*, *i*)

leo.external.leoSAGlobals.**getPythonEncodingFromString**(*s*)
Return the encoding given by Python's encoding line. s is the entire file.

leo.external.leoSAGlobals.**getWord**(*s*, *i*)
Return i,j such that s[i:j] is the word surrounding s[i].

leo.external.leoSAGlobals.**get_leading_ws**(*s*)
Returns the leading whitespace of 's'.

leo.external.leoSAGlobals.**get_line**(*s*, *i*)

leo.external.leoSAGlobals.**get_line_after**(*s*, *i*)

leo.external.leoSAGlobals.**isBytes**(*s*)
Return True if s is Python3k bytes type.

leo.external.leoSAGlobals.**isCallable**(*obj*)

leo.external.leoSAGlobals.**isChar**(*s*)
Return True if s is a Python2K character type.

leo.external.leoSAGlobals.**isMacOS**()

leo.external.leoSAGlobals.**isString**(*s*)
Return True if s is any string, but not bytes.

leo.external.leoSAGlobals.**isUnicode**(*s*)
Return True if s is a unicode string.

leo.external.leoSAGlobals.**isValidEncoding**(*encoding*)

leo.external.leoSAGlobals.**isWordChar**(*ch*)
Return True if ch should be considered a letter.

leo.external.leoSAGlobals.**isWordChar1**(*ch*)

leo.external.leoSAGlobals.**is_c_id**(*ch*)

leo.external.leoSAGlobals.**is_nl**(*s*, *i*)

leo.external.leoSAGlobals.**is_special**(*s*, *i*, *directive*)
Return True if the body text contains the @ directive.

leo.external.leoSAGlobals.**is_ws**(*c*)

leo.external.leoSAGlobals.**is_ws_or_nl**(*s*, *i*)

leo.external.leoSAGlobals.**joinLines**(*aList*)

leo.external.leoSAGlobals.**joinlines**(*aList*)

leo.external.leoSAGlobals.**listToString**(*aList*, *tag=None*, *sort=False*, *indent=''*, *toRepr=False*)

leo.external.leoSAGlobals.**makeDict**(***keys*)
Returns a Python dictionary from using the optional keyword arguments.

leo.external.leoSAGlobals.**match**(*s*, *i*, *pattern*)

leo.external.leoSAGlobals.**match_c_word**(*s*, *i*, *name*)

leo.external.leoSAGlobals.**match_ignoring_case**(*s1*, *s2*)

leo.external.leoSAGlobals.**match_word**(*s*, *i*, *pattern*)

leo.external.leoSAGlobals.**maxStringListLength**(*aList*)
Return the maximum string length in a list of strings.

leo.external.leoSAGlobals.**note**(**args*, ***keys*)

class leo.external.leoSAGlobals.**nullObject**(**args*, ***keys*)
An object that does nothing, and does it very well.

leo.external.leoSAGlobals.**oldCheckVersion**(*version*, *againstVersion*, *condition='>='*, *stringCompare='0.0.0'*, *delimiter='.'*)

leo.external.leoSAGlobals.**optimizeLeadingWhitespace**(*line*, *tab_width*)

leo.external.leoSAGlobals.**os_path_abspath**(*path*)
Convert a path to an absolute path.

leo.external.leoSAGlobals.**os_path_basename**(*path*)
Return the second half of the pair returned by split(path).

leo.external.leoSAGlobals.**os_path_dirname**(*path*)
Return the first half of the pair returned by split(path).

leo.external.leoSAGlobals.**os_path_exists**(*path*)
Return True if path exists.

leo.external.leoSAGlobals.**os_path_expandExpression**(*s*, ***keys*)
Expand {{anExpression}} in c's context.

leo.external.leoSAGlobals.**os_path_expanduser**(*path*)
wrap os.path.expanduser

leo.external.leoSAGlobals.**os_path_finalize**(*path*, ***keys*)
Expand '~', then return os.path.normpath, os.path.abspath of the path.

There is no corresponding os.path method

leo.external.leoSAGlobals.**os_path_finalize_join**(**args*, ***keys*)
Do os.path.join(*args), then finalize the result.

leo.external.leoSAGlobals.**os_path_getmtime**(*path*)
Return the modification time of path.

leo.external.leoSAGlobals.**os_path_getsize**(*path*)
Return the size of path.

leo.external.leoSAGlobals.**os_path_isabs**(*path*)
Return True if path is an absolute path.

leo.external.leoSAGlobals.**os_path_isdir**(*path*)
Return True if the path is a directory.

leo.external.leoSAGlobals.**os_path_isfile**(*path*)
Return True if path is a file.

leo.external.leoSAGlobals.**os_path_join**(**args*, ***keys*)

leo.external.leoSAGlobals.**os_path_normcase**(*path*)
Normalize the path's case.

leo.external.leoSAGlobals.**os_path_normpath**(*path*)
Normalize the path.

leo.external.leoSAGlobals.**os_path_realpath**(*path*)

leo.external.leoSAGlobals.**os_path_split**(*path*)

leo.external.leoSAGlobals.**os_path_splitext**(*path*)

leo.external.leoSAGlobals.**os_startfile**(*fname*)

leo.external.leoSAGlobals.**pause**(*s*)

leo.external.leoSAGlobals.**pdb**(*message=''*)
Fall into pdb.

leo.external.leoSAGlobals.**pr**(**args*, ***keys*)
Print all non-keyword args, and put them to the log pane. The first, third, fifth, etc. arg translated by translateString. Supports color, comma, newline, spaces and tabName keyword arguments.

leo.external.leoSAGlobals.**prettyPrintType**(*obj*)

leo.external.leoSAGlobals.**printDict**(*d*, *tag=''*, *verbose=True*, *indent=''*)

leo.external.leoSAGlobals.**printList**(*aList*, *tag=None*, *sort=False*, *indent=''*)

leo.external.leoSAGlobals.**printStack**()

leo.external.leoSAGlobals.**print_dict**(*d*, *tag=”*, *verbose=True*, *indent=”*)

leo.external.leoSAGlobals.**print_list**(*aList*, *tag=None*, *sort=False*, *indent=”*)

leo.external.leoSAGlobals.**print_obj**(*obj*, *tag=None*, *sort=False*, *verbose=True*, *indent=”*)

leo.external.leoSAGlobals.**print_stack**()

leo.external.leoSAGlobals.**removeBlankLines**(*s*)

leo.external.leoSAGlobals.**removeExtraLws**(*s*, *tab_width*)
Remove extra indentation from one or more lines.

Warning: used by getScript. This is *not* the same as adjustTripleString.

leo.external.leoSAGlobals.**removeLeading**(*s*, *chars*)
Remove all characters in chars from the front of s.

leo.external.leoSAGlobals.**removeLeadingBlankLines**(*s*)

leo.external.leoSAGlobals.**removeLeadingWhitespace**(*s*, *first_ws*, *tab_width*)

leo.external.leoSAGlobals.**removeTrailing**(*s*, *chars*)
Remove all characters in chars from the end of s.

leo.external.leoSAGlobals.**removeTrailingWs**(*s*)

leo.external.leoSAGlobals.**reportBadChars**(*s*, *encoding*)

leo.external.leoSAGlobals.**scanf**(*s*, *pat*)

leo.external.leoSAGlobals.**shortFileName**(*fileName*)

leo.external.leoSAGlobals.**shortFilename**(*fileName*)

leo.external.leoSAGlobals.**skip_blank_lines**(*s*, *i*)

leo.external.leoSAGlobals.**skip_c_id**(*s*, *i*)

leo.external.leoSAGlobals.**skip_id**(*s*, *i*, *chars=None*)

leo.external.leoSAGlobals.**skip_leading_ws**(*s*, *i*, *ws*, *tab_width*)

leo.external.leoSAGlobals.**skip_leading_ws_with_indent**(*s*, *i*, *tab_width*)
Skips leading whitespace and returns (i, indent),

- i points after the whitespace
- indent is the width of the whitespace, assuming tab_width wide tabs.

leo.external.leoSAGlobals.**skip_line**(*s*, *i*)

leo.external.leoSAGlobals.**skip_long**(*s*, *i*)
Scan s[i:] for a valid int. Return (i, val) or (i, None) if s[i] does not point at a number.

leo.external.leoSAGlobals.**skip_matching_c_delims**(*s*, *i*, *delim1*, *delim2*, *reverse=False*)
Skip from the opening delim to the matching delim2.

Return the index of the matching ‘)’, or -1

leo.external.leoSAGlobals.**skip_matching_python_delims**(*s*, *i*, *delim1*, *delim2*, *reverse=False*)
Skip from the opening delim to the matching delim2.

Return the index of the matching ‘)’, or -1

leo.external.leoSAGlobals.**skip_matching_python_parens**(*s*, *i*)
Skip from the opening (to the matching).

Return the index of the matching ‘)’, or -1

leo.external.leoSAGlobals.**skip_nl**(*s*, *i*)
Skips a single “logical” end-of-line character.

leo.external.leoSAGlobals.**skip_non_ws**(*s*, *i*)

leo.external.leoSAGlobals.**skip_pascal_braces**(*s*, *i*)

leo.external.leoSAGlobals.**skip_to_char**(*s*, *i*, *ch*)

leo.external.leoSAGlobals.**skip_to_end_of_line**(*s*, *i*)

leo.external.leoSAGlobals.**skip_to_start_of_line**(*s*, *i*)

leo.external.leoSAGlobals.**skip_ws**(*s*, *i*)

leo.external.leoSAGlobals.**skip_ws_and_nl**(*s*, *i*)

leo.external.leoSAGlobals.**splitLines**(*s*)
Split s into lines, preserving the number of lines and the endings of all lines, including the last line.

leo.external.leoSAGlobals.**splitlines**(*s*)
Split s into lines, preserving the number of lines and the endings of all lines, including the last line.

leo.external.leoSAGlobals.**stripBlankLines**(*s*)

leo.external.leoSAGlobals.**stripBrackets**(*s*)
Same as s.lstrip(‘<’).rstrip(‘>’) except it works for Python 2.2.1.

leo.external.leoSAGlobals.**toEncodedString**(*s*, *encoding='utf-8'*, *reportErrors=False*)

leo.external.leoSAGlobals.**toGuiIndex**(*s*, *index*)
Convert index to a Python int.

index may be a Tk index (x.y) or ‘end’.

leo.external.leoSAGlobals.**toPythonIndex**(*s*, *index*)
Convert index to a Python int.

index may be a Tk index (x.y) or ‘end’.

leo.external.leoSAGlobals.**toString**(*obj*, *tag=None*, *sort=False*, *verbose=True*, *indent=''*)

leo.external.leoSAGlobals.**toUnicode**(*s*, *encoding='utf-8'*, *reportErrors=False*)

leo.external.leoSAGlobals.**toUnicodeFileEncoding**(*path*)

leo.external.leoSAGlobals.**toUnicodeWithErrorCode**(*s*, *encoding*, *reportErrors=False*)

leo.external.leoSAGlobals.**tr**(*s*)
Return the translated text of s.

leo.external.leoSAGlobals.**trace**(**args*, ***keys*)

leo.external.leoSAGlobals.**translateArgs**(*args*, *d*)
Return the concatenation of s and all args,

with odd args translated.

leo.external.leoSAGlobals.**translateString**(*s*)
Return the translated text of s.

leo.external.leoSAGlobals.**u**(*s*)

leo.external.leoSAGlobals.**ue**(*s*, *encoding*)

leo.external.leoSAGlobals.**virtual_event_name**(*s*)

leo.external.leoSAGlobals.**warning**(**args*, ***keys*)

1.2.3.6 leoftsindex Module

1.2.3.7 leosax Module

Read .leo files into a simple python data structure with h, b, u (unknown attribs), gnx and children information. Clones and derived files are ignored. Useful for scanning multiple .leo files quickly.

class leo.external.leosax.**LeoNode**
Bases: object

Representation of a Leo node. Root node has itself as parent.

IVariables

children python list of children

u unknownAttributes dict (decoded)

h headline

b body text

gnx node id

parent node's parent

path list of nodes that lead to this one from root, including this one

UNL()
Return the UNL string leading to this node

flat()
iterate this node and all its descendants in a flat list, useful for finding things and building an UNL based view

node_pos_count(*node*)
node_pos_count - return the position (index) and count of preceeding siblings with the same name, also return headline

Parameters **node** (LeoNode) – node to characterize

Returns h, pos, count

Return type (str, int, int)

class leo.external.leosax.**LeoReader**(**args*, ***kwargs*)
Bases: xml.sax.handler.ContentHandler

Read .leo files into a simple python data structure with h, b, u (unknown attribs), gnx and children information. Clones and derived files are ignored. Useful for scanning multiple .leo files quickly.

IVariables

root root node

cur used internally during SAX read

idx mapping from gnx to node

in_ name of XML element we're current in, used for SAX read

in_attr attributes of element tag we're currentl in, used for SAX read

path list of nodes leading to current node

characters(*content*)
collect body text and headlines

endElement(*name*)
decode unknownAttributes when t element is done

startElement(*name*, *attrs*)
collect information from v and t elements

leo.external.leosax.**get_leo_data**(*source*)
Return the root node for the specificed .leo file (path or file)

1.2.3.8 lproto Module

1.2.3.9 stringlist Module

class leo.external.stringlist.**SList**
Bases: list

List derivative with a special access attributes.

These are normal string lists, but with the special attributes:

.l: value as list (the list itself). .n: value as a string, joined on newlines. .s: value as a string, joined on spaces.

fields(**fields*)
Collect whitespace-separated fields from string list

Allows quick awk-like usage of string lists.

Example data (in var a, created by 'a = !ls -l')::

-rwxrwxrwx 1 ville None 18 Dec 14 2006 ChangeLog

drwxrwxrwx+ 6 ville None 0 Oct 24 18:05 IPython

a.fields(0) is ['-rwxrwxrwx', 'drwxrwxrwx+'] a.fields(1,0) is ['1 -rwxrwxrwx', '6 drwxrwxrwx+'] (note the joining by space). a.fields(-1) is ['ChangeLog', 'IPython']

IndexErrors are ignored.

Without args, fields() just split()'s the strings.

get_list()

get_nlstr()

get_spstr()

grep(*pattern*, *prune=False*, *field=None*)
Return all strings matching 'pattern' (a regex or callable)

This is case-insensitive. If prune is true, return all items NOT matching the pattern.

If field is specified, the match must occur in the specified whitespace-separated field.

Examples:

```
a.grep( lambda x: x.startswith('C') )
a.grep('Cha.*log', prune=1)
a.grep('chm', field=-1)
```

l

n

s

sort (*field=None*, *nums=False*)
sort by specified fields (see fields())

Example:: a.sort(1, nums = True)

Sorts a by second field, in numerical order (so that 21 > 3)

leo.external.stringlist.**shcmd**(*cmd*)
Execute shell command, capture output to string list

1.2.3.10 Subpackages

concurrent Package

concurrent Package

Subpackages

futures Package

futures Package

Execute computations asynchronously using threads or processes.

_base Module

exception leo.external.concurrent.futures._base.**CancelledError**
Bases: *leo.external.concurrent.futures._base.Error*

The Future was cancelled.

class leo.external.concurrent.futures._base.**DoneAndNotDoneFutures**(*done*, *not_done*)
Bases: tuple

done
Alias for field number 0

not_done
Alias for field number 1

exception leo.external.concurrent.futures._base.**Error**
Bases: exceptions.Exception

Base class for all future-related exceptions.

class leo.external.concurrent.futures._base.**Executor**
Bases: object

This is an abstract base class for concrete asynchronous executors.

map(*fn*, **iterables*, ***kwargs*)
Returns a iterator equivalent to map(fn, iter).

Args:

fn: A callable that will take take as many arguments as there are passed iterables.

timeout: The maximum number of seconds to wait. If None, then there is no limit on the wait time.

Returns: An iterator equivalent to: map(func, *iterables) but the calls may be evaluated out-of-order.

Raises:

TimeoutError: If the entire result iterator could not be generated before the given timeout.

Exception: If fn(*args) raises for any values.

shutdown(*wait=True*)
Clean-up the resources associated with the Executor.

It is safe to call this method several times. Otherwise, no other methods can be called after this one.

Args:

wait: If True then shutdown will not return until all running futures have finished executing and the resources used by the executor have been reclaimed.

submit(*fn*, **args*, ***kwargs*)
Submits a callable to be executed with the given arguments.

Schedules the callable to be executed as fn(*args, **kwargs) and returns a Future instance representing the execution of the callable.

Returns: A Future representing the given call.

class leo.external.concurrent.futures._base.**Future**
Bases: object

Represents the result of an asynchronous computation.

add_done_callback(*fn*)
Attaches a callable that will be called when the future finishes.

Args:

fn: A callable that will be called with this future as its only argument when the future completes or is cancelled. The callable will always be called by a thread in the same process in which it was added. If the future has already completed or been cancelled then the callable will be called immediately. These callables are called in the order that they were added.

cancel()
Cancel the future if possible.

Returns True if the future was cancelled, False otherwise. A future cannot be cancelled if it is running or has already completed.

cancelled()
Return True if the future has cancelled.

done()
Return True of the future was cancelled or finished executing.

exception(*timeout=None*)
Return the exception raised by the call that the future represents.

Args:

timeout: The number of seconds to wait for the exception if the future isn't done. If None, then there is no limit on the wait time.

Returns: The exception raised by the call that the future represents or None if the call completed without raising.

Raises: CancelledError: If the future was cancelled. TimeoutError: If the future didn't finish executing before the given

timeout.

result(*timeout=None*)
Return the result of the call that the future represents.

Args:

timeout: The number of seconds to wait for the result if the future isn't done. If None, then there is no limit on the wait time.

Returns: The result of the call that the future represents.

Raises: CancelledError: If the future was cancelled. TimeoutError: If the future didn't finish executing before the given

timeout.

Exception: If the call raised then that exception will be raised.

running()
Return True if the future is currently executing.

set_exception(*exception*)
Sets the result of the future as being the given exception.

Should only be used by Executor implementations and unit tests.

set_result(*result*)
Sets the return value of work associated with the future.

Should only be used by Executor implementations and unit tests.

set_running_or_notify_cancel()
Mark the future as running or process any cancel notifications.

Should only be used by Executor implementations and unit tests.

If the future has been cancelled (cancel() was called and returned True) then any threads waiting on the future completing (though calls to as_completed() or wait()) are notified and False is returned.

If the future was not cancelled then it is put in the running state (future calls to running() will return True) and True is returned.

This method should be called by Executor implementations before executing the work associated with this future. If this method returns False then the work should not be executed.

Returns: False if the Future was cancelled, True otherwise.

Raises:

RuntimeError: if this method was already called or if set_result() or set_exception() was called.

exception leo.external.concurrent.futures._base.**TimeoutError**
Bases: *leo.external.concurrent.futures._base.Error*

The operation exceeded the given deadline.

leo.external.concurrent.futures._base.**as_completed**(*fs*, *timeout=None*)
An iterator over the given futures that yields each as it completes.

Args:

fs: The sequence of Futures (possibly created by different Executors) to iterate over.

timeout: The maximum number of seconds to wait. If None, then there is no limit on the wait time.

Returns: An iterator that yields the given Futures as they complete (finished or cancelled).

Raises:

TimeoutError: If the entire result iterator could not be generated before the given timeout.

leo.external.concurrent.futures._base.**wait**(*fs*, *timeout=None*, *return_when='ALL_COMPLETED'*)
Wait for the futures in the given sequence to complete.

Args:

fs: The sequence of Futures (possibly created by different Executors) to wait upon.

timeout: The maximum number of seconds to wait. If None, then there is no limit on the wait time.

return_when: Indicates when this function should return. The options are:

FIRST_COMPLETED - Return when any future finishes or is cancelled.

FIRST_EXCEPTION - Return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL_COMPLETED.

ALL_COMPLETED - Return when all futures finish or are cancelled.

Returns: A named 2-tuple of sets. The first set, named 'done', contains the futures that completed (is finished or cancelled) before the wait completed. The second set, named 'not_done', contains uncompleted futures.

_compat Module

leo.external.concurrent.futures._compat.**namedtuple**(*typename*, *field_names*)
Returns a new subclass of tuple with named fields.

```
>>> Point = namedtuple('Point', 'x y')
>>> Point.__doc__                   # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)             # instantiate with positional args or keywords
>>> p[0] + p[1]                     # indexable like a plain tuple
33
>>> x, y = p                        # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                       # fields also accessable by name
33
>>> d = p._asdict()                 # convert to a dictionary
>>> d['x']
11
```

(continues on next page)

(continued from previous page)

```
>>> Point(**d)                             # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100)                      # _replace() is like str.replace() but␣
↪targets named fields
Point(x=100, y=22)
```

process Module

Implements ProcessPoolExecutor.

The follow diagram and text describe the data-flow through the system:

|======================= In-process ====================|== Out-of-process ==|

+———-+ +———-+ +———+ +———+ +———+ | | => | Work Ids | => | | => | Call Q | => | | | | +———+ | | +———+ | | | | | . . . | | | | . . . | | | | | | 6 | | | | 5, call() | | | | | | 7 | | | | . . . | | | | Process | | . . . | | Local | +———+ | Process | | Pool | +———-+ | Worker | | #1..n | | Executor | | Thread | | | | | +——— + | | +———+ | | | | <=> | Work Items | <=> | | <= | Result Q | <= | | | | +———+ | | +———+ | | | | | 6: call() | | | | . . . | | | | | | future | | | | 4, result | | | | | | . . . | | | | 3, except | | | +———-+ +———+ +———+ +———+ +———+

Executor.submit() called: - creates a uniquely numbered _WorkItem and adds it to the “Work Items” dict - adds the id of the _WorkItem to the “Work Ids” queue

Local worker thread: - reads work ids from the “Work Ids” queue and looks up the corresponding

> WorkItem from the “Work Items” dict: if the work item has been cancelled then it is simply removed from the dict, otherwise it is repackaged as a _CallItem and put in the “Call Q”. New _CallItems are put in the “Call Q” until “Call Q” is full. NOTE: the size of the “Call Q” is kept small because calls placed in the “Call Q” can no longer be cancelled with Future.cancel().

- reads _ResultItems from “Result Q”, updates the future stored in the “Work Items” dict and deletes the dict entry

Process #1..n: - reads _CallItems from “Call Q”, executes the calls, and puts the resulting

> _ResultItems in “Request Q”

class leo.external.concurrent.futures.process.**ProcessPoolExecutor**(*max_workers=None*)

Bases: concurrent.futures._base.Executor

shutdown(*wait=True*)

Clean-up the resources associated with the Executor.

It is safe to call this method several times. Otherwise, no other methods can be called after this one.

Args:

wait: If True then shutdown will not return until all running futures have finished executing and the resources used by the executor have been reclaimed.

submit(*fn*, **args*, ***kwargs*)

Submits a callable to be executed with the given arguments.

Schedules the callable to be executed as fn(*args, **kwargs) and returns a Future instance representing the execution of the callable.

Returns: A Future representing the given call.

thread Module

Implements ThreadPoolExecutor.

class leo.external.concurrent.futures.thread.**ThreadPoolExecutor**(*max_workers*)
Bases: concurrent.futures._base.Executor

shutdown(*wait=True*)
Clean-up the resources associated with the Executor.

It is safe to call this method several times. Otherwise, no other methods can be called after this one.

Args:

wait: If True then shutdown will not return until all running futures have finished executing and the resources used by the executor have been reclaimed.

submit(*fn*, **args*, ***kwargs*)
Submits a callable to be executed with the given arguments.

Schedules the callable to be executed as fn(*args, **kwargs) and returns a Future instance representing the execution of the callable.

Returns: A Future representing the given call.

1.2.4 plugins Package

1.2.4.1 plugins Package

1.2.4.2 FileActions Module

Defines actions taken when double-clicking on @<file> nodes and supports @file-ref nodes.

The double-click-icon-box command on 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.

leo.plugins.FileActions.**applyFileAction**(*p*, *filename*, *c*)

leo.plugins.FileActions.**doFileAction**(*filename*, *c*)

leo.plugins.FileActions.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.FileActions.**onIconDoubleClick**(*tag*, *keywords*)

leo.plugins.FileActions.**shellScriptInWindow**(*c*, *script*)

1.2.4.3 active_path Module

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 you double-click the node.

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 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.

@auto nodes can be set up for existing files can be loaded by double clicking on the node. 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
- pick dir - select a folder interactively to make a new top level @path node
- mark-content - mark outline content in the @path tree, as opposed to filesystem content. Useful if you want to delete the @path tree to check for content not on the filesystem first

If you want to use an input other than double clicking a node set active_path_event to a value like 'hypercclick1' or 'headrclick1'.

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).

Autoloading can be toggled with *active-path-toggle-autoload*, autoloading defaults to initially on unless @bool active-path-do-autoload = False.

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.

Per Folder file/folder inclusion and exclusion by adding flags to the body of an active path folder (either `@` or `/*/`), can include multiple `inc=` and `exc=` flags:

- `excdirs` - excludes all directories
- `excfiles` - excludes all files
- `inc=` - a single item or comma separated list of strings to include in the list of files/folders
- `exc=` - a single item or comma separated list of strings to exclude in the list of files/folders
- `re` - search using regular expressions (otherwise a case-sensitive 'in' comparison)

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.

leo.plugins.active_path.**active_path_act_on_node**(*event*, *p=None*)
act_on_node handler for active_path.py

leo.plugins.active_path.**attachToCommander**(*t*, *k*)

leo.plugins.active_path.**checkIncExc**(*item*, *inc*, *exc*, *regEx*)
Primary logic to check if an item is in either the include or exclude list

leo.plugins.active_path.**cmd_ActOnNode**(*event*, *p=None*)
act_on_node handler for active_path.py

leo.plugins.active_path.**cmd_LoadRecursive**(*event*)
Recursive update, with expansions.

leo.plugins.active_path.**cmd_MakeDir**(*event*)

leo.plugins.active_path.**cmd_MarkContent**(*event*)
cmd_MarkContent - mark nodes in @path sub-tree with non-filesystem content

i.e. not organizer nodes (no body), subdirs (body starts with @path) vanished file placeholders, or @<file> nodes.

leo.plugins.active_path.**cmd_PickDir**(*event*)
cmd_PickDir - Show user a folder picker to create

leo.plugins.active_path.**cmd_PurgeUnloadedFilesHere**(*event*)
Remove files never loaded, i.e. no kind of @file node.

leo.plugins.active_path.**cmd_PurgeUnloadedFilesRecursive**(*event*)
Remove files never loaded, i.e. no kind of @file node.

leo.plugins.active_path.**cmd_PurgeVanishedFilesHere**(*event*)
Remove files no longer present, i.e. "*filename*" entries.

leo.plugins.active_path.**cmd_PurgeVanishedFilesRecursive**(*event*)
Remove files no longer present, i.e. "*filename*" entries.

leo.plugins.active_path.**cmd_SetNodeToAbsolutePath**(*event*, *p=None*)
Change "/dirname/" to "@path /absolute/path/to/dirname".

leo.plugins.active_path.**cmd_SetNodeToAbsolutePathRecursive**(*event*)
Change "/dirname/" to "@path /absolute/path/to/dirname", recursively

leo.plugins.active_path.**cmd_ShowCurrentPath**(*event*)
Just show the path to the current file/directory node in the log pane.

leo.plugins.active_path.**cmd_ToggleAutoLoad**(*event*)
cmd_ToggleAutoLoad - toggle autoloading behavior

leo.plugins.active_path.**cmd_UpdateRecursive**(*event*)
Recursive update, no new expansions.

leo.plugins.active_path.**cond**(*p*)

leo.plugins.active_path.**condunl**(*p*)

leo.plugins.active_path.**createDir**(*c*, *parent*, *d*)
Ask if we should create a new folder

leo.plugins.active_path.**createFile**(*c*, *parent*, *d*)
Ask if we should create a new file

leo.plugins.active_path.**deleteChildren**(*p*, *cond*, *dtor=None*)

leo.plugins.active_path.**deleteDescendents**(*p*, *cond*, *dtor=None*, *descendAnyway=False*, *_culls=0*)

leo.plugins.active_path.**deleteTestHierachy**(*c*)

leo.plugins.active_path.**dtor**(*p*)

leo.plugins.active_path.**flattenOrganizers**(*p*)
Children of p, some of which may be in organizer nodes

In the following example nodeA's children are nodes B, F, and G:

```
/nodeA/
   nodeB
   /nodeC/
      nodeD
      nodeE
   oldStuff
      nodeF
      nodeG
```

leo.plugins.active_path.**getPath**(*c*, *p*)

leo.plugins.active_path.**getPathOld**(*p*)

leo.plugins.active_path.**inAny**(*item*, *group*, *regEx=False*)
Helper function to check if word from list is in a string

leo.plugins.active_path.**inReList**(*txt*, *lst*)

leo.plugins.active_path.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.active_path.**isDirNode**(*p*)

leo.plugins.active_path.**isFileNode**(*p*)
really isEligibleToBecomeAFileNode

leo.plugins.active_path.**loadDocstring**(*file_path*)

leo.plugins.active_path.**makeTestHierachy**(*c*)

leo.plugins.active_path.**onSelect**(*tag*, *keywords*)
Determine if a file or directory node was clicked, and the path

leo.plugins.active_path.**openDir**(*c*, *parent*, *d*)
Expand / refresh an existing folder

Note: With the addition of per folder inclusion/exclusion a check is done against both the current list of nodes and against the files/folders as they exist on the system. This check must be done in both places to keep the node list in sync with the file system while respecting the inc/exc lists - John Lunzer

leo.plugins.active_path.**openFile**(*c*, *parent*, *d*, *autoload=False*)
Open an existing file

leo.plugins.active_path.**popup_entry**(*c*, *p*, *menu*)
Populate the Path submenu of the popup.

leo.plugins.active_path.**query**(*c*, *s*)
Return yes/no answer from user for question s

leo.plugins.active_path.**run_recursive**(*c*)
Recursive descent.

leo.plugins.active_path.**subDir**(*d*, *p*)

leo.plugins.active_path.**sync_node_to_folder**(*c*, *parent*, *d*, *updateOnly=False*, *recurse=False*)
Decide whether we're opening or creating a file or a folder

1.2.4.4 add_directives Module

Allows users to define new @direcives.

leo.plugins.add_directives.**addPluginDirectives**(*tag*, *keywords*)
Add all new directives to g.globalDirectiveList

leo.plugins.add_directives.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.add_directives.**scanPluginDirectives**(*tag*, *keywords*)
Add a tuple (d,v,s,k) to list for every directive d found

1.2.4.5 at_folder Module

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 (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.

leo.plugins.at_folder.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.at_folder.**onSelect**(*tag*, *keywords*)

leo.plugins.at_folder.**sync_node_to_folder**(*c*, *parent*, *d*)

1.2.4.6 at_produce Module

Executes commands in nodes whose body text starts with @produce.

WARNING: trying to execute a non-existent command will hang Leo.

To use, put in the body text of a node:

```
@produce echo hi
```

This plugin creates two new commands: at-produce-all and at-produce-selected.

at-produce-all scans the entire tree for body text containing @produce. at-produce-selected just scans the selected tree.

Whatever follows @produce is executed as a command.

@produce commands are executed in the order they are found, that is, in outline order.

The at-produce commands produce a log node as the last top-level node of the outline. Any output, including error messages, should be there.

This plugin is not intended as a replacement for make or Ant, but as a simple substitute when that machinery is overkill.

leo.plugins.at_produce.**addMenu**(*tag*, *keywords*)
Produce two new entries at the end of the Outlines menu.

leo.plugins.at_produce.**getList**(*c*, *all*)
Return a list of all @produce lines in body texts in an outline. all = True: scan c's entire outline. all = False: scan c.p and its descendants.

leo.plugins.at_produce.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.at_produce.**produce_all_f**(*event*)

leo.plugins.at_produce.**produce_selected_f**(*event*)

leo.plugins.at_produce.**run**(*c*, *all*)
Run all @produce nodes in a separate thread. Report progress via a timer in *this* thread.

leo.plugins.at_produce.**runList**(*c*, *aList*)
Run all commands in aList (in a separate thread). Do not change Leo's outline in this thread!

leo.plugins.at_produce.**timer_callback_helper**(*c*, *t*, *timer*)
All drawing must be done in the main thread.

1.2.4.7 at_view Module

1.2.4.8 attrib_edit Module

1.2.4.9 backlink Module

1.2.4.10 baseNativeTree Module

1.2.4.11 bibtex Module

Creates a BibTex file from an '@bibtex <filename>' tree.

Nodes of the form '@<x> key' create entries in the file.

When the user creates a new node (presses enter in headline text) the plugin automatically inserts a template for the entry in the body pane.

The 'templates' dict in the << globals >> section defines the template. The default, the template creates all required entries.

Double-clicking the @bibtex node writes the file. For example, the following outline:

```
-@bibtex biblio.bib
 +@book key,
  author = {A. Uthor},
  year = 1999
```

creates the following 'biblio.bib' file:

```
@book{key,
author = {A. Uthor},
year= 1999}
```

@string nodes define strings and may contain multiple entries. The plugin writes all @string nodes at the start of the file. For example, the following outline:

```
-@bibtext biblio.bib
 +@string
  j1 = {Journal1}
 +@article AUj1
  author = {A. Uthor},
  journal = j1
 +@string
  j2 = {Journal2}
  j3 = {Journal3}
```

creates the following file:

```
@string{j1 = {Journal1}}
@string{j2 = {Journal2}}
@string{j3 = {Journal3}}

@article{AUj1,
author = {A. Uthor},
journal = j1}
```

Headlines that do not start with '@' are organizer nodes: the plugin does not write organizer nodes, but does write descendant nodes.

BibTeX files can be imported by creating an empty node with '@bibtex filename' in the headline. Double-clicking it will read the file and parse it into a @bibtex tree. No syntax checks are made: the file is expected to be a valid BibTeX file.

`leo.plugins.bibtex.`**`init`**`()`
Return True if the plugin has loaded successfully.

`leo.plugins.bibtex.`**`onHeadKey`**`(`*tag*, *keywords*`)`
Write template for the entry in body pane.

If body pane is empty, get template for the entry from a dictionary 'templates ' and write it in the body pane.

20141127 - note headkey2 now only fires on *Enter*, no need to check which key brought us here.

`leo.plugins.bibtex.`**`onIconDoubleClick`**`(`*tag*, *keywords*`)`
Read or write a bibtex file when the node is double-clicked.

Write the @bibtex tree as bibtex file when the root node is double-clicked. If it has no child nodes, read bibtex file.

leo.plugins.bibtex.**readBibTexFileIntoTree**(*c*, *fn*, *p*)
Import a BibTeX file into a @bibtex tree.

leo.plugins.bibtex.**writeTreeAsBibTex**(*c*, *fn*, *root*)
Write root's *subtree* to bibFile.

1.2.4.12 bigdash Module

1.2.4.13 bookmarks Module

1.2.4.14 bzr_qcommands Module

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.

leo.plugins.bzr_qcommands.**bzr_qcommands**(*c*, *p*, *menu*)
see module docs.

leo.plugins.bzr_qcommands.**init**()
Return True if the plugin has loaded successfully.

1.2.4.15 chapter_hoist Module

1.2.4.16 codewisecompleter Module

1.2.4.17 colorize_headlines Module

Manipulates appearance of individual tree widget items. (Qt only).

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.

leo.plugins.colorize_headlines.**colorize_headlines_visitor**(*c*, *p*, *item*)
Changes @thin, @auto, @shadow to bold

leo.plugins.colorize_headlines.**init**()
Return True if the plugin has loaded successfully.

1.2.4.18 contextmenu Module

1.2.4.19 ctagscompleter Module

1.2.4.20 cursesGui Module

1.2.4.21 datenodes Module

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

class leo.plugins.datenodes.**DateNodes**(*c*)
Bases: object

Main DateNodes class

ascii_encoder()

boolean_settings = ['datenodes_month_node_omit_saturdays', 'datenodes_month_node_omit_

default_settings = {'datenodes_body_text': 'To do...', 'datenodes_day_node_headline':

insert_day_node(*event=None*)

insert_month_node(*event=None*)

insert_year_node(*event=None*)

leo.plugins.datenodes.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.datenodes.**on_create**(*tag*, *keywords*)

1.2.4.22 debugger_pudb Module

Makes g.pdb() enter the Pudb debugger instead of pdb.

Pudb is a full-screen Python debugger: http://pypi.python.org/pypi/pudb

leo.plugins.debugger_pudb.**init**()
Return True if the plugin has loaded successfully.

1.2.4.23 dragdropgoodies Module

1.2.4.24 dtest Module

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
```

class leo.plugins.dtest.**DT**(*tag*, *keywords*)

Bases: *leo.core.leoPlugins.BaseLeoPlugin*

Sends code to the doctest module and reports the result If text is selected, tests only the selection.

```
>>> print("hello world")
hello world
>>> g.es('hello world')
>>> print(c.p.h)
Docstring
>>> import notfound
Traceback (most recent call last):
    ...
ImportError: No module named notfound
>>>
```

dtest(*event*)

The handler for dtest

leo.plugins.dtest.**init**()

Return True if the plugin has loaded successfully.

1.2.4.25 dump_globals Module

Dumps Python globals at startup.

leo.plugins.dump_globals.**init**()

Return True if the plugin has loaded successfully.

leo.plugins.dump_globals.**onStart**(*tag*, *keywords*)

1.2.4.26 empty_leo_file Module

Allows Leo to open any empty file as a minimal .leo file.

leo.plugins.empty_leo_file.**init**()

Return True if the plugin has loaded successfully.

leo.plugins.empty_leo_file.**onOpen**(*tag*, *keywords*)

1.2.4.27 enable_gc Module

Enables debugging and tracing for Python's garbage collector.

leo.plugins.enable_gc.**init**()

Return True if the plugin has loaded successfully.

leo.plugins.enable_gc.**onStart**(*tag*, *keywords*)

1.2.4.28 expfolder Module

Adds @expfolder nodes that represent folders in the file system.

The double-click-icon-box command on an @expfolder node 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.

leo.plugins.expfolder.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.expfolder.**on_icondclick**(*tag*, *keywords*)

1.2.4.29 free_layout Module

1.2.4.30 ftp Module

1.2.4.31 geotag Module

Tags nodes with latitude and longitude.

leo.plugins.geotag.**cmd_OpenServerPage**(*event*)

leo.plugins.geotag.**cmd_ShowNode**(*event*)

leo.plugins.geotag.**cmd_TagNode**(*event*)

class leo.plugins.geotag.**geotag_Controller**(*c*)
Bases: object

A per-commander class that manages geotagging.

callback(*data*)

static getAttr(*p*)

leo.plugins.geotag.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.geotag.**onCreate**(*tag*, *key*)

leo.plugins.geotag.**onQuit**(*tag*, *key*)

1.2.4.32 gitarchive Module

Store snapshots of outline in git.

leo.plugins.gitarchive.**contfile**(*c*, *p*)

leo.plugins.gitarchive.**git_dump_f**(*event*)

leo.plugins.gitarchive.**git_log_f**(*event*)

leo.plugins.gitarchive.**init**()
Return True if the plugin has loaded successfully.

1.2.4.33 graphcanvas Module

1.2.4.34 gtkDialogs Module

1.2.4.35 gtkGui Module

1.2.4.36 import_cisco_config Module

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.

leo.plugins.import_cisco_config.**create_import_cisco_menu**(*tag*, *keywords*)

leo.plugins.import_cisco_config.**importCiscoConfig**(*c*)

leo.plugins.import_cisco_config.**init**()
Return True if the plugin has loaded successfully.

1.2.4.37 initinclass Module

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.plugins.initinclass.**InitInClass**(*tag*, *keywords*)
Move __init__ into the class node body in python @auto imports

leo.plugins.initinclass.**init**()
Return True if the plugin has loaded successfully.

1.2.4.38 interact Module

1.2.4.39 internal_ipkernel Module

1.2.4.40 ipython Module

1.2.4.41 ironPythonGui Module

1.2.4.42 jinjarender Module

Render @jinja nodes.

- sudo apt-get install python-jinja2

Create headline like this:

@jinja ~/foo.txt

Select the node and do alt-x act-on-node

Conceptually, acts like @nosent - tree is parsed, template is expanded and content is written to the file.

Requires "valuespace" plugin. Fetches vars from valuespace.

class leo.plugins.jinjarender.**JinjaCl**(*c*)
Bases: object

leo.plugins.jinjarender.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.jinjarender.**jinja_act_on_node**(*c*, *p*, *event*)

leo.plugins.jinjarender.**jinja_install**()

leo.plugins.jinjarender.**jinja_render**(*template*, *fname*, *d*)

leo.plugins.jinjarender.**untangle**(*c*, *p*)

1.2.4.43 leoOPML Module

A plugin to read and write Leo outlines in .opml (http://en.wikipedia.org/wiki/OPML) format.

The OPML plugin creates two new commands that read and write Leo outlines in OPML format. The read-opml-file command creates a Leo outline from an .opml file. The write-opml-file command writes the present Leo outline to an .opml file.

Various settings control what gets written to .opml files, and in what format. As usual, you specify settings for the OPML plugin using leoSettings.leo. The settings for the OPML are found in the node: @settings–>Plugins–>opml plugin.

Here are the settings that control the format of .opml files. The default values are shown.

- @string opml_namespace = leo:com:leo-opml-version-1

The namespace urn for the xmlns attribute of <opml> elements. This value typically is not used, but it should refer to Leo in some way.

- @bool opml_use_outline_elements = True
- If True, Leo writes body text to <leo:body> elements nested in <outline> elements. Otherwise, Leo writes body text to leo:body attributes of <outline> elements.
- @string opml_version = 2.0

The opml version string written to the <OPML> element. Use 2.0 unless there is a specific reason to use 1.0.

- @bool opml_write_body_text = True

Leo writes body text to the OPML file only if this is True.

- @bool opml_write_leo_details = True

If True, Leo writes the native attributes of Leo's <v> elements as attributes of the opml <outline> elements.

The native attributes of <v> elements are a, t, vtag (new), tnodeList, marks, expanded and descendentTnodeUnknownAttributes.

- @bool opml_write_leo_globals_attributes = True

If True, Leo writes body_outline_ratio‘ and global_window_position attributes to the <head> element of the .opml file.

- @bool opml_write_ua_attributes

If True, write unknownAttributes **NOTE**: ua_attributes are not currently read from opml.

- @bool opml_expand_ua_dictionary

If True, expand an unknownAttriubte ‘x’ of type dict to ‘ua_x_key0’, ‘ua_x_key1’ etc. **WARNING**: using this feature may prevent reading these ua_attributes from opml, if that feature is implemented in the future.

- @bool opml_skip_ua_dictionary_blanks

If True, when expanding as above, skip blank dict entries.

class leo.plugins.leoOPML.**NodeClass**
Bases: object

A class representing one outline element.

Use getters to access the attributes, properties and rules of this mode.

dump()

class leo.plugins.leoOPML.**OpmlController**(*c*)
Bases: object

The controller class for this plugin.

cleanSaxInputString(*s*)
Clean control characters from s. s may be a bytes or a (unicode) string.

createChildren(*c*, *node*, *parent_v*)

createTnodesDict()
Create c.tnodesDict by from self.generated_gnxs by converting VNode entries to tnodes.

createVnode(*c*, *node*, *v=None*)

createVnodes(*c*, *dummyRoot*)
Important: this method and its helpers are low-level code corresponding to link/unlink methods in leoNodes.py. Modify this with extreme care.

dumpTree(*root*, *dummy=True*)

handleVnodeAttributes(*node*, *v*)

parse_opml_file(*fn*)

putToOPML(*owner*)
Write the c.p as OPML, using the owner's put method.

readFile(*fileName*)
Read the opml file.

readOpmlCommand(*event=None*)
Open a Leo window containing the contents of an .opml file.

reloadSettings()

resolveTnodeLists(*c*)

setCurrentPosition(*c*)

writeFile(*fileName*)
Write fileName as an OPML file.

writeOpmlCommand(*event=None*)
Save a Leo outline to an OPMLfile.

class leo.plugins.leoOPML.**PutToOPML**(*owner*)
Bases: object

Write c.p's tree as OPML, using the owner's put method.

aAttributes(*p*)

attributeEscape(*s*)

initConfig()
Init all configuration settings.

put(*s*)

putAll()
Put the selected outline as OPML. All elements and attributes prefixed by 'leo:' are leo-specific. All other elements and attributes are specified by the OPML 1 spec.

putOPMLHeader()
Put the OPML header, including attributes for globals, prefs and find settings.

putOPMLNode(*p*)

putOPMLNodes()

putOPMLPostlog()

putOPMLProlog()

putXMLLine()
Put the **properly encoded** <?xml> element.

tnodeListAttributes(*p*)
Put the tnodeList attribute of p.v

uAAttributes(*p*)
write unknownAttributes with various levels of expansion

class leo.plugins.leoOPML.**SaxContentHandler**(*c*, *inputFileName*)
Bases: xml.sax.saxutils.XMLGenerator

A sax content handler class that reads OPML files.

attrsToList(*attrs*)
Convert the attributes to a list of g.Bunches. attrs: an Attributes item passed to startElement.

attrsToString(*attrs*, *sep='\n'*)
Convert the attributes to a string.

attrs: an Attributes item passed to startElement.

sep: the separator charater between attributes.

characters(*content*)

clean(*s*)

define_dispatch_dict()

doGlobalWindowAttributes(*attrs*)

doHeadAttributes(*attrs*)

doOutlineAttributes(*attrs*)

endBodyText()
End a <leo:body> element.

endDocument()

endElement(*name*)

endElementNS(*name*, *qname*)

endOutline()

error(*message*)

getNode()

ignorableWhitespace(*content*)

inElement(*name*)

printStartElement(*name*, *attrs*)

processingInstruction(*target*, *data*)

skippedEntity(*name*)

startBodyText(*attrs*)
Start a <leo:body> element.

startDocument()

startElement(*name*, *attrs*)

startElementNS(*name*, *qname*, *attrs*)

startHead(*attrs*)

startOutline(*attrs*)

startWinPos(*attrs*)

leo.plugins.leoOPML.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.leoOPML.**onCreate**(*tag*, *keys*)

1.2.4.44 leo_interface Module

Allows the user to browse XML documents in Leo.

This plugin implements an interface to XML generation, so that the resulting file can be processed by leo.

class leo_file represents the whole leo file. class leo_node has a headline and body text.

See the end of this file for a minimal example on how to use these classes.

If you encounter the first of a set of clones, create a leo_node. If you encounter the same set of clones later, create a leo_clone node and refer back to the first element.

class leo.plugins.leo_interface.**LeoNode**
Bases: object

Abstrace class for generating xml.

add_child(*child*)

gen(*file*)

gen_children(*file*)

mark(*file*, *marker*, *func*, *newline=True*)

mark_with_attributes(*file*, *marker*, *attribute_list*, *func*, *newline=True*)

mark_with_attributes_short(*file*, *marker*, *attribute_list*)

nthChild(*n*)

leo.plugins.leo_interface.**escape**(*s*)

leo.plugins.leo_interface.**init**()
Return True if the plugin has loaded successfully.

class leo.plugins.leo_interface.**leo_clone**(*orig*)
Bases: *leo.plugins.leo_interface.node_with_parent*

Class representing a clone.

The (only) data of a clone is the reference to a leo_node.

When you encounter the first clone of a set of clones, generate a leo_node. The second clone should then reference this leo_node, and contain no other data.

Since clones are indistinguishable, there is really not much to do in this class.

gen_tnodes(*file*)

gen_vnodes(*file*)

class leo.plugins.leo_interface.**leo_file**
Bases: *leo.plugins.leo_interface.LeoNode*

Leo specific class representing a file.

empty(*file*)

find_panel_settings(*file*)

gen(*file*)

gen1(*file*)

gen_tnodes(*file*)

gen_vnodes(*file*)

headString()

header(*file*)

max_tnode_index()

nr_tnodes()

parent()

preferences(*file*)

sss(*file*)

class leo.plugins.leo_interface.**leo_node**(*headline=''*, *body=''*)
Bases: *leo.plugins.leo_interface.LeoNode*, *leo.plugins.leo_interface.node_with_parent*

Leo specific class representing a node.

These nodes correspond to tnodes in LEO. They have a headline and a body.

They also represent the (only) vnode in an outline without clones.

bodyString(*body*)

count = 0

gen_tnodes(*file*)

gen_tnodes1(*file*)

gen_vnodes(*file*)

gen_vnodes1(*file*)

headString()

set_body(*body*)

set_headline(*headline*)

write_body_escaped(*file*)

write_headline(*file*)

write_headline_escaped(*file*)

leo.plugins.leo_interface.**leotree**()

class leo.plugins.leo_interface.**node_with_parent**
Bases: object

parent()

set_parent(*node*)

1.2.4.45 leo_pdf Module

This NOT a Leo plugin: this is a docutils writer for .pdf files.

That file uses the reportlab module to convert html markup to pdf.

The original code written by Engelbert Gruber.

Rewritten by Edward K. Ream for the Leo rst3 plugin.

class leo.plugins.leo_pdf.**Bunch**(***keywords*)
Bases: object

A class that represents a colection of things.

Especially useful for representing a collection of related variables.

get(*key*, *theDefault=None*)

ivars()

keys()

toString()

class leo.plugins.leo_pdf.**PDFTranslator**(*writer*, *doctree*)
Bases: docutils.nodes.NodeVisitor

as_what()

createParagraph(*text*, *style='Normal'*, *bulletText=None*)

depart_Text(*node*)

depart_address(*node*)

depart_admonition()

depart_attention(*node*)

depart_author(*node*)

depart_authors(*node*)

depart_block_quote(*node*)

depart_bullet_list(*node*)

depart_caption(*node*)

depart_caution(*node*)

depart_citation(*node*)

depart_citation_reference(*node*)

depart_classifier(*node*)

depart_colspec(*node*)

depart_comment(*node*)
Invisible nodes should be ignored.

depart_contact(*node*)

depart_copyright(*node*)

depart_danger(*node*)

depart_date(*node*)

depart_definition(*node*)

depart_definition_list(*node*)

depart_definition_list_item(*node*)

depart_description(*node*)

depart_docinfo(*node*)

depart_docinfo_item()

depart_doctest_block(*node*)

depart_document(*node*)

depart_emphasis(*node*)

depart_entry(*node*)

depart_enumerated_list(*node*)

depart_error(*node*)

depart_field(*node*)

depart_field_argument(*node*)

depart_field_body(*node*)

depart_field_list(*node*)

depart_field_name(*node*)

depart_figure(*node*)
Invisible nodes should be ignored.

depart_footnote(*node*)

depart_footnote_reference(*node*)

depart_generated(*node*)

depart_hint(*node*)

depart_image(*node*)

depart_important(*node*)

depart_interpreted(*node*)

depart_label(*node*)

depart_legend(*node*)

depart_line_block(*node*)

depart_list_item(*node*)

depart_literal(*node*)

depart_literal_block(*node*)

depart_meta(*node*)

depart_note(*node*)

depart_option(*node*)

depart_option_argument(*node*)

depart_option_group(*node*)

depart_option_list(*node*)

depart_option_list_item(*node*)

depart_option_string(*node*)

depart_organization(*node*)

depart_paragraph(*node*)

depart_pending(*node*)
Invisible nodes should be ignored.

depart_problematic(*node*)

depart_reference(*node*)

depart_revision(*node*)

depart_row(*node*)

depart_section(*node*)

depart_sidebar(*node*)
Invisible nodes should be ignored.

depart_status(*node*)

depart_strong(*node*)

depart_substitution_definition(*node*)
Invisible nodes should be ignored.

depart_subtitle(*node*)

depart_system_message(*node*)

depart_table(*node*)
Invisible nodes should be ignored.

depart_target(*node*)

depart_tbody(*node*)
Invisible nodes should be ignored.

depart_term(*node*)

depart_tgroup(*node*)
Invisible nodes should be ignored.

depart_thead(*node*)
Invisible nodes should be ignored.

depart_tip(*node*)
Invisible nodes should be ignored.

depart_title(*node*)

depart_title_reference(*node*)
Invisible nodes should be ignored.

depart_topic(*node*)

depart_transition(*node*)
Invisible nodes should be ignored.

depart_version(*node*)

depart_warning(*node*)
Invisible nodes should be ignored.

dumpContext()

dumpNode(*node*, *tag=''*)

encode(*text*)
Encode special characters in *text* & return.

escape(*s*)

footnote_backrefs(*node*)
Create b.link and b.setLink for visit/depart_label.

footnote_backrefs_depart(*node*)

inContext(*kind*)
Return the most recent bunch having the indicated kind, or None.

invisible_visit(*node*)
Invisible nodes should be ignored.

pdfMunge(*s*)
Duplicate the munging done (somewhere in docutils) of section names.

This allows us to use the nameids attribute in the document element.

peek(*kind*)

pop(*kind*)

push(***keys*)

putHead(*start*, *style='Normal'*, *bulletText=None*)

putTail(*start*, *style='Normal'*, *bulletText=None*)

starttag(*node*, *tagname*, *suffix='\n'*, *caller=''*, ***attributes*)

unimplemented_visit(*node*)

visit_Text(*node*)

visit_address(*node*)

visit_admonition(*node*, *name*)

visit_attention(*node*)

visit_author(*node*)

visit_authors(*node*)

visit_block_quote(*node*)

visit_bullet_list(*node*)

visit_caption(*node*)

visit_caution(*node*)

visit_citation(*node*)

visit_citation_reference(*node*)

visit_classifier(*node*)

visit_colspec(*node*)

visit_comment(*node*)

visit_contact(*node*)

visit_copyright(*node*)

visit_danger(*node*)

visit_date(*node*)

visit_definition(*node*)

visit_definition_list(*node*)

visit_definition_list_item(*node*)

visit_description(*node*)

visit_docinfo(*node*)

visit_docinfo_item(*node*, *name*)

visit_doctest_block(*node*)

visit_document(*node*)

visit_emphasis(*node*)

visit_entry(*node*)

visit_enumerated_list(*node*)

visit_error(*node*)

visit_field(*node*)

visit_field_argument(*node*)

visit_field_body(*node*)

visit_field_list(*node*)

visit_field_name(*node*)

visit_figure(*node*)

visit_footnote(*node*)

visit_footnote_reference(*node*)
Generate code for a footnote reference.

visit_generated(*node*)

visit_hint(*node*)

visit_image(*node*)

visit_important(*node*)

visit_interpreted(*node*)

visit_label(*node*)

visit_legend(*node*)

visit_line_block(*node*)

visit_list_item(*node*)

visit_literal(*node*)

visit_literal_block(*node*)

visit_meta(*node*)

visit_note(*node*)

visit_option(*node*)

visit_option_argument(*node*)

visit_option_group(*node*)

visit_option_list(*node*)

visit_option_list_item(*node*)

visit_option_string(*node*)

visit_organization(*node*)

visit_paragraph(*node*)

visit_pending(*node*)
Invisible nodes should be ignored.

visit_problematic(*node*)

visit_raw(*node*)

visit_reference(*node*)

visit_revision(*node*)

visit_row(*node*)

visit_section(*node*)

visit_sidebar(*node*)
Invisible nodes should be ignored.

visit_status(*node*)

visit_strong(*node*)

visit_substitution_definition(*node*)

visit_subtitle(*node*)

visit_system_message(*node*)

visit_table(*node*)
Invisible nodes should be ignored.

visit_target(*node*)

visit_tbody(*node*)
Invisible nodes should be ignored.

visit_term(*node*)

visit_tgroup(*node*)
Invisible nodes should be ignored.

visit_thead(*node*)
Invisible nodes should be ignored.

visit_tip(*node*)
Invisible nodes should be ignored.

visit_title(*node*)

visit_title_reference(*node*)
Invisible nodes should be ignored.

visit_topic(*node*)

visit_transition(*node*)
Invisible nodes should be ignored.

visit_version(*node*)

visit_warning(*node*)
Invisible nodes should be ignored.

class leo.plugins.leo_pdf.**Writer**
Bases: docutils.writers.Writer

createPDF_usingPlatypus(*story*)

createParagraphsFromIntermediateFile(*s*, *story*, *visitor*)

lower()

output = None

settings_spec = ('PDF-Specific Options', None, (('Specify a stylesheet URL, used verba

supported = ('pdf', 'rlpdf')

translate()
Do final translation of self.document into self.output.

leo.plugins.leo_pdf.**bunch**
alias of *leo.plugins.leo_pdf.Bunch*

class leo.plugins.leo_pdf.**dummyPDFTranslator**(*writer*, *doctree*, *contents*)
Bases: docutils.nodes.NodeVisitor

as_what()

buildFromIntermediateFile()
Synthesize calls to reportlab.platypus.para.Paragraph from an intermediate file.

depart_document(*node*)

encode(*text*)
Encode special characters in *text* & return.

putParaFromIntermediateFile(*lines*, *style*)

visit_document(*node*)

leo.plugins.leo_pdf.**getStyleSheet**()
Returns a stylesheet object

leo.plugins.leo_pdf.**get_language**(*doctree*)
A wrapper for changing docutils get_language method.

leo.plugins.leo_pdf.**init**()
This file may be distributed in Leo's plugin folder, but this file is NOT a Leo plugin!

The init method returns False to tell Leo's plugin manager and unit tests to skip this file.

1.2.4.46 leo_to_html Module

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

Note: As of Mar. 2014 regular Leo @string settings starting with *leo_to_html_* are checked first, before the `.ini` file. E.g. `@string leo_to_html_flagjustheadlines = No` has the same effect as `flagjustheadlines = No` in the `.ini`, and takes precedence.

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.

class leo.plugins.leo_to_html.**Leo_to_HTML**(*c=None*)
Bases: object

This class provides all the functionality of the leo_to_html plugin.

See the docstring for the leo_to_html module for details.

announce(*msg*, *prefix=None*, *color=None*, *silent=None*)
Print a message if flags allow.

announce_end(*msg='done'*, *prefix=None*, *color=None*)

announce_fail(*msg='failed'*, *prefix=None*, *color=None*)

announce_start(*msg='running ...'*, *prefix=None*, *color=None*)

applyTemplate(*template=None*)
Fit self.xhtml and self.title into an (x)html template.

Plaace the result in self.xhtml.

The template string in self.template should have too %s place holders. The first for the title the second for the body.

doBodyElement(*pp*, *level=None*)
Append wrapped body string to output stream.

doHeadline(*p*, *level=None*)
Append wrapped headstring to output stream.

doItemBulletList(*p*)
" Recursivley proccess an outline node into an xhtml list.

doItemHeadlineTags(*p*, *level=1*)
" Recursivley proccess an outline node into an xhtml list.

do_xhtml(*node=False*)
Convert the tree to xhtml.

Return the result as a string in self.xhtml.

Only the code to represent the tree is generated, not the wraper code to turn it into a file.

getPlainTemplate()
Returns a string containing a template for the outline page.

The string should have positions in order, for: title and body text.

getXHTMLTemplate()
Returns a string containing a template for the outline page.

The string should have positions in order, for: title and body text.

loadConfig()
Load configuration from a .ini file.

main(*bullet=None*, *show=False*, *node=False*)
Generate the html and write the files.

If 'bullet' is not recognized then the value of bullet_type from the the properties file will be used.

If 'show' is True then the file will be saved to a temp dir and shown in a browser.

setup()
Set various parameters.

show()
Convert the outline to xhtml and display the results in a browser.

If browser_command is set, this command will be used to launch the browser. If it is not set, or if the command fails, the default browser will be used. Setting browser_command to a bad command will slow down browser launch.

showSubtree(*p*)
Return True if subtree should be shown.

subtree should be shown if it is not an @file node or if it is an @file node and flags say it should be shown.

write(*name*, *data*, *basedir=None*, *path=None*)
Write a single file.

The *name* can be a file name or a ralative path which will be added to basedir and path to create a full path for the file to be written.

If basedir is None self.basedir will be used and if path is none self.path will be used.

writeall()
Write all the files

leo.plugins.leo_to_html.**abspath**(**args*)
Join the arguments and convert to an absolute file path.

leo.plugins.leo_to_html.**createExportMenus**(*tag*, *keywords*)
Create menu items in File -> Export menu.

Menu's will not be created if the following appears in an @setting tree:

```
@bool leo_to_html_no_menus = True
```

This is so that the user can use @menu to decide which commands will appear in the menu and where.

leo.plugins.leo_to_html.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.leo_to_html.**onCreate**(*tag*, *keys*)
Handle 'after-create-leo-frame' hooks by creating a plugin controller for the commander issuing the hook.

class leo.plugins.leo_to_html.**pluginController**(*c*)
Bases: object

A per commander plugin controller to create and handle minibuffer commands that control the plugins functions.

export_html(*event=None*, *bullet=None*, *show=False*, *node=False*)
Command handler for leo_to_html. See modules docstring for details.

export_html_bullet(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

export_html_head(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

export_html_node(*event=None*, *bullet=None*)
Command handler for leo_to_html. See modules docstring for details.

export_html_node_bullet(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

export_html_node_head(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

export_html_node_number(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

export_html_number(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html(*event=None*, *bullet=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_bullet(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_head(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_node(*event=None*, *bullet=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_node_bullet(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_node_head(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_node_number(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

show_html_number(*event=None*)
Command handler for leo_to_html. See modules docstring for details.

leo.plugins.leo_to_html.**safe**(*s*)
Convert special characters to html entities.

1.2.4.47 leo_to_rtf Module

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.

This plug-in loads installs an "Outline to Microsoft RTF" menu item in 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.

leo.plugins.leo_to_rtf.**createExportMenu**(*tag*, *keywords*)

leo.plugins.leo_to_rtf.**export_rtf**(*c*)

leo.plugins.leo_to_rtf.**init**()
Return True if the plugin has loaded successfully.

1.2.4.48 leocursor Module

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.

class leo.plugins.leocursor.**AM_CapColon**
Bases: *leo.plugins.leocursor.AM_Colon*

Like AM_Colon, but first letter must be capital.

pattern = '^([A-Z][A-Za-z0-9_]*:)(\\s+(\\S.*))*$'

class leo.plugins.leocursor.**AM_Colon**
Bases: *leo.plugins.leocursor.AttribManager*

Attributes are in the body text as:

```
start-of-line letter letters-or-numbers colon space(s) attribute-value
```

Both:

```
foo:
```

and:

```
foo: all this is the value
```

are attributes (first one is value==''), but:

```
foo:value
```

is not (because there's no space after the colon).

filterBody(*b*)
Return the body string without any parts used to store attributes, if this flavor of attribute manager stores attributes in the body. If not, just return the whole body string.

getAttrib(*v*, *what*)
Get an attribute value from a vnode

has_key(*v*, *what*)

keys(*v*)
Get list of attribute keys from a vnode

pattern = <_sre.SRE_Pattern object>

class leo.plugins.leocursor.**AttribManager**
Bases: object

Class responsible for reading / writing attributes from vnodes for LeoCursor

exception NotPresent
Bases: exceptions.Exception

filterBody(*b*)
Return the body string without any parts used to store attributes, if this flavor of attribute manager stores attributes in the body. If not, just return the whole body string.

getAttrib(*v*, *what*)
Get an attribute value from a vnode

keys(*v*)
Get list of attribute keys from a vnode

class leo.plugins.leocursor.**LeoCursor**(*v*, *other=None*)
Bases: object

See module docs.

exception NotPresent
Bases: exceptions.Exception

1.2.4.49 leofeeds Module

Read feeds from rss / atom / whatever sources

Usage: Create node with a headline like:

@feed http://www.planetqt.org/atom.xml

(or somesuch).

Do alt-x act-on-node on that node to populate the subtree from the feed data. Requires "feedparser" python module

class leo.plugins.leofeeds.**MLStripper**
Bases: HTMLParser.HTMLParser

get_fed_data()

handle_data(*data*)

leo.plugins.leofeeds.**chi**(*p*)

leo.plugins.leofeeds.**emit**(*r*, *h*, *b*)

leo.plugins.leofeeds.**emitfeed**(*url*, *p*)

leo.plugins.leofeeds.**feeds_act_on_node**(*c*, *p*, *event*)

leo.plugins.leofeeds.**feeds_install**()

leo.plugins.leofeeds.**init**()

leo.plugins.leofeeds.**onCreate**(*tag*, *keys*)

leo.plugins.leofeeds.**strip_tags**(*cont*)

1.2.4.50 leofts Module

1.2.4.51 leomail Module

Sync local mailbox files over to Leo.

Creates mail-refresh command, which can only be applied to @mbox nodes of the form:

> @mbox <path to .mbox file>

The command parses the .mbox file and creates a separate node for each thread.

Replies to the original messages become children of that message.

class leo.plugins.leomail.**MLStripper**
Bases: HTMLParser.HTMLParser

get_data()

handle_data(*data*)

leo.plugins.leomail.**emit_message**(*c*, *parent*, *root*, *message*)
Create all the children of p.

leo.plugins.leomail.**init**()

leo.plugins.leomail.**mail_refresh**(*event*)

leo.plugins.leomail.**strip_tags**(*obj*)

1.2.4.52 leomylyn Module

Provides an experience like Mylyn:http://en.wikipedia.org/wiki/Mylyn for Leo.

It "scores" the nodes based on how interesting they probably are for you, allowing you to focus on your "working set".

Scoring is based on how much you edit the nodes.

class leo.plugins.leomylyn.**MylynController**
Bases: object

add_score(*v*, *points*)

children_hnd(*tag*, *kw*)

content_hnd(*tag*, *kw*)

set_handlers()

leo.plugins.leomylyn.**init**()
Return True if the plugin has loaded successfully.

1.2.4.53 leoremote Module

1.2.4.54 leoscreen Module

1.2.4.55 lineNumbers Module

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.

leo.plugins.lineNumbers.**init**()
Return True if the plugin has loaded successfully.

1.2.4.56 livecode Module

1.2.4.57 macros Module

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!

class leo.plugins.macros.**ParamClass**(*c*)
Bases: object

addMenu()
Add a submenu in the outline menu.

findParameters(*p*)
Find the parameterized nodes in p's parents..

parameterize(*event=None*)

leo.plugins.macros.**init**()
Return True if this plugin loaded correctly.

leo.plugins.macros.**onCreate**(*tag*, *keywords*)
Create the per-commander instance of ParamClass.

1.2.4.58 maximizeNewWindows Module

Maximizes all new windows.

leo.plugins.maximizeNewWindows.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.maximizeNewWindows.**maximize_window**(*tag*, *keywords*)

1.2.4.59 mime Module

Opens files with their default platform program.

The double-click-icon-box command on @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. For 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.

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.

leo.plugins.mime.**exec_full_cmd**(*cmd*)
Accept a command string including filename and return a function which executes the command.

leo.plugins.mime.**exec_string_cmd**(*cmd*)
Accept a command string and return a function which opens executes the command, replacing %s with the full file path.

leo.plugins.mime.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.mime.**open_mimetype**(*tag*, *keywords*, *val=None*)
Simulate double-clicking on the filename in a file manager. Order of preference is:

1. @string mime_open_cmd setting
2. _mime_open_cmd, defined per sys.platform detection
3. open_func(fpath), defined per sys.platform detection
4. mailcap file for mimetype handling

1.2.4.60 mnplugins Module

mnplugins.py

mnplugins shows how to : define new Commands "insertOK" + "insertUser" create Usermenu with new Commands

new Commands: insertOK:

insert 'OK' in headline and a stamp in the first body line are there child nodes without 'OK' verhindern OK in actual node. The right-click-icon command also inserts 'OK'.

insertUser [Shift-F6] insert a <user/date/time> stamp at the current location in body text

leo.plugins.mnplugins.**create_UserMenu**(*tag*, *keywords*)

leo.plugins.mnplugins.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.mnplugins.**insertBodystamp**(*c*, *v*)

leo.plugins.mnplugins.**insertOKcmd**(*self*, *event=None*)

leo.plugins.mnplugins.**insertUser**(*self*, *event=None*)
Handle the Insert User command.

leo.plugins.mnplugins.**is_subnodesOK**(*v*)

leo.plugins.mnplugins.**mnOKstamp**()

leo.plugins.mnplugins.**mnstamp**()

leo.plugins.mnplugins.**onRclick**(*tag*, *keywords*)
Handle right click in body pane.

leo.plugins.mnplugins.**onStart**(*tag*, *keywords*)

leo.plugins.mnplugins.**setHeadOK**(*c*, *v*)

1.2.4.61 mod_autosave Module

1.2.4.62 mod_framesize Module

Sets a hardcoded frame size.

Prevents Leo from setting custom frame size (e.g. from an external .leo document)

leo.plugins.mod_framesize.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.mod_framesize.**setTopGeometry_mod_framesize**(*self*, **args*)
Monkeypatced version of setTopGeometry

1.2.4.63 mod_http Module

An http plug-in for LEO, based on AsyncHttpServer.py.

Adapted and extended from the Python Cookbook: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/259148

This plug-in has three distinct behaviors:

- Viewing loaded outlines in a browser, e.g. at http://localhost:8130/
- Storing web bookmarks in Leo outlines from a browser, e.g. at http://localhost:8130/_/add/bkmk/
- Remotely executing code in a running Leo instance, e.g. at http://localhost:8130/_/exec/?cmd=nd=p.insertAsLastChild()

Install this plug-in is as follows:

Start Leo with the plug-in enabled. You will see a purple message that says something like:

```
"http serving enabled at 127.0.0.1:8130"
```

Settings

@bool http_active = True required for plug-in to be active

@string http_ip = 127.0.0.1 address to bind to, see notes below

@int http_port = 8130 port to use (1 3 0 ~= L E O)

@bool http_allow_remote_exec = False must be changed to True for remote code execution

@string rst_http_attributename = 'rst_http_attribute' link to obsolete rst3 plugin

@data user_bookmark_stylesheet Additional .css for bookmarks.

@data http_stylesheet The default .css for this page.

@data user_http_stylesheet Additional .css for this page.

Note: The html generated by the server contains both stylesheets as inline <style> elements, with the user_http_stylesheet contents last.

@data mod_http script The body text of this @data setting contains *all* the javascript used in the page.

Note The html generated by the server handles the <script> elements:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
</script>
<script>..the contents of @data mod_http_script..
</script>
```

Browsing Leo files

Start a web browser, and enter the following URL: http://localhost:8130/

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.

Note: IP address 127.0.0.1 is accessible by all users logged into your local machine. That means while Leo and mod_http is running anyone logged into your machine will be able to browse all your leo outlines and add bookmarks.

Note: If you want all other network accessible machines to have access to your mod_http instance, then use @string http_ip = 0.0.0.0.

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.

Saving bookmarks from browser to Leo

To do this, add a bookmark to the browser with the following URL / Location:

```
javascript:w=window; d=w.document; ln=[];if(w.location.href.indexOf('one-tab')>-1)
↪{el=d.querySelectorAll('a');for (i in el){ln.push({url:el[i].href,txt:el[i].
↪innerHTML});};};w.open('http://localhost:8130/_/add/bkmk/?&name=' + escape(d.title)␣
↪+ '&selection=' + escape(window.getSelection()) + '&ln=' + escape(JSON.
↪stringify(ln)) + '&url=' + escape(w.location.href),"_blank","toolbar=no,␣
↪location=no, directories=no, status=no, menubar=no, scrollbars=no, resizable=yes,␣
↪copyhistory=no, width=800, height=300, status=no");void(0); # NOQA
```

and edit the port (8130 in the example above) to match the port you're using for mod_http.

Bookmarks are created as the first node in the outline which has been opened longest. You can set the `@string http_bookmark_unl` to specify an alternative location, e.g.:

```
@string http_bookmark_unl = /home/tbrown/.bookmarks.leo#@bookmarks-->Incoming
```

to place them in the *Incoming* node in the *@bookmarks* node in the *.bookmarks.leo* outline.

The headline is preceded with '@url ' *unless* the `bookmarks` plug-in is loaded. If the `bookmarks` plug-in is loaded the bookmark will have to be moved to a `@bookmarks` tree to be useful.

Note: there is special support for Chrome's OneTab extension as a mechanism for saving all tabs open. Click the OneTab button to get a list of tabs, then click the "Share all as web page" link to show the list on www.one-tab.com. Bookmark as usual as described above. You can then delete the shared list with the "Delete this shared page" button. The Leo node created will have a child node for each of the listed tabs.

The browser may or may not be able to close the bookmark form window for you, depending on settings - set `dom.allow_scripts_to_close_windows` to true in `about:config` in Firefox.

Executing code remotely

Warning:

> Allowing remote code execution is a **HUGE SECURITY HOLE**, you need to be sure that the url from which you access Leo (typically http://localhost:8130/) is accessible only by people and software you trust.

Remote execution is turned off by default, you need to manually / locally change the @setting `@bool http_allow_remote_exec = False` to `True` to enable it.

Commands to be executed are submitted via HTTP GET requests, which can be generated in almost any language and also triggered from shortcuts, links in other documents or applications, etc. etc.

The basic form is:

```
http://localhost:8130/_/exec/?cmd=<python code for Leo to execute>
```

The query parameters are:

cmd (required) A valid python snippet for Leo to execute. Executed by the `eval` command in the `mod_scripting` plug-in. Can be specified multiple times, each is executed in order. May contain newlines, see examples.

c (optional) Which currently loaded outline to use, can be an integer, starting from zero, or the full path+filename, or just the base filename. Defaults to 0 (zero), i.e. the "first" open outline.

enc (optional) Encoding for response, 'str', 'repr', or 'json'. Used to render the returned value.

mime_type (optional) Defaults to `text/plain`. Could be useful to use `text/html` etc.

A special variant url is:

```
http://localhost:8130/_/exec/commanders/
```

which returns a list of open outlines.

Examples

This command:

```
curl http://localhost:8130/_/exec/?cmd='c.bringToFront()' >/dev/null
```

will raise the Leo window, or at least make the window manager signal the need to raise it.

```
curl --get --data-urlencode       cmd='g.handleUrl("file:///home/tbrown/.leo/.
↪contacts.leo#Contacts", c)'       http://localhost:8130/_/exec/ >/dev/null
```

will cause a running Leo instance to open `/some/path/contacts.leo` and select the `Contacts` node. A desktop icon link, browser bookmark, or link in a spread-sheet or other document could be used the same way.

In the `bash` shell language, this code:

```
TEXT="$@"
curl --silent --show-error --get --data-urlencode cmd="
    nd = c.rootPosition().insertAfter()
    nd.h = 'TODO: $TEXT'
    import time
    nd.b = '# created %s' % time.asctime()
    c.selectPosition(nd)
    c.redraw()
    'To do item created
```

‘ ” http://localhost:8130/_/exec/

could be written in a file called `td`, and then, assuming that file is executable and on the shell's path, entering:

```
td remember to vacuum the cat
```

on the command line would create a node at the top of the first open outline in Leo with a headline `TODO: remember to vacuum the cat` and a body text `# created Wed Jul 29 16:42:26 2015`. The command `vs-eval` returns the value of the last expression in a block, so the trailing `'To do item created '` gives better feedback than `None` generated by `c.redraw()`. `c.selectPosition(nd)` is important ant to stop Leo getting confused about which node is selected.

class leo.plugins.mod_http.**ExecHandler**(*request_handler*)
Bases: object

Quasi-RPC GET based interface

get_response()
Return the file like 'f' that leo_interface.send_head makes

proc_cmds()

class leo.plugins.mod_http.**LeoActions**(*request_handler*)
Bases: object

A place to collect other URL based actions like saving bookmarks from the browser. Conceptually this stuff could go in class leo_interface but putting it here for separation for now.

add_bookmark()
Return the file like 'f' that leo_interface.send_head makes

add_bookmark_selection(*node*, *text*)
Insert the selected text into the bookmark node, after any earlier selections but before the users comments.

http://example.com/

Tags: tags, are here

Full title of the page

Collected: timestamp

""" The first saved selection """

""" The second saved selection """

Users comments

i.e. just above the "Users comments" line.

get_favicon()

get_one_tab(*links*, *nd*)
get_one_tab - Add child bookmarks from OneTab chrome extension

Parameters

- *links*: list of {'txt':, 'url':} dicts
- *nd*: node under which to put child nodes

get_response()
Return the file like 'f' that leo_interface.send_head makes

class leo.plugins.mod_http.**RequestHandler**(*conn*, *addr*, *server*)
Bases: *leo.plugins.mod_http.leo_interface*, asynchat.async_chat, SimpleHTTPServer.SimpleHTTPRequestHandler

collect_incoming_data(*data*)
Collects the data arriving on the connexion

copyfile(*source*, *outputfile*)
Copy all data between two file objects.

The SOURCE argument is a file object open for reading (or anything with a read() method) and the DESTINATION argument is a file object open for writing (or anything with a write() method).

The only reason for overriding this would be to change the block size or perhaps to replace newlines by CRLF – note however that this the default server uses this to copy binary data as well.

do_GET()
Begins serving a GET request

do_POST()
Begins serving a POST request. The request data must be readable on a file-like object called self.rfile

finish()
Reset terminator (required after POST method), then close

found_terminator()

handle_data()
Class to override

handle_post_data()
Called when a POST request body has been read

handle_read_event()
Over-ride SimpleHTTPRequestHandler.handle_read_event.

handle_request_line()
Called when the http request line and headers have been received

log_message(*format*, **args*)
Log an arbitrary message.

This is used by all other logging functions. Override it if you have specific logging wishes.

The first argument, FORMAT, is a format string for the message to be logged. If the format string contains any % escapes requiring parameters, they should be specified as subsequent arguments (it's just like printf!).

The client host and current date/time are prefixed to every message.

prepare_POST()
Prepare to read the request body

query(*parsedQuery*)
Returns the QUERY dictionary, similar to the result of cgi.parse_qs except that : - if the key ends with [], returns the value (a Python list) - if not, returns a string, empty if the list is empty, or with the first value in the list

class leo.plugins.mod_http.**Server**(*ip*, *port*, *handler*)
Bases: asyncore.dispatcher

Copied from http_server in medusa

handle_accept()

leo.plugins.mod_http.**a_read**(*obj*)

class leo.plugins.mod_http.**config**
Bases: object

http_active = False

http_ip = '127.0.0.1'

http_port = 8130

http_timeout = 0

rst2_http_attributename = 'rst_http_attribute'

class leo.plugins.mod_http.**delayedSocketStream**(*sock*)
Bases: asyncore.dispatcher_with_send

handle_read()

initiate_sending()

writable()

write(*data*)

leo.plugins.mod_http.**escape**(*s*)

leo.plugins.mod_http.**getConfiguration**(*c*)
Called when the user opens a new file.

leo.plugins.mod_http.**getData**(*setting*)
Return the given @data node.

leo.plugins.mod_http.**getGlobalConfiguration**()
read config.

leo.plugins.mod_http.**get_http_attribute**(*p*)

leo.plugins.mod_http.**init**()
Return True if the plugin has loaded successfully.

class leo.plugins.mod_http.**leo_interface**
Bases: object

add_leo_links(*window*, *node*, *f*)

Given a node 'node', add links to: The next sibling, if any. the next node. the parent. The children, if any.

create_href(*href*, *text*, *f*)

create_leo_h_reference(*window*, *node*)

create_leo_reference(*window*, *node*, *text*, *f*)
Create a reference to 'node' in 'window', displaying 'text'

find_window_and_root(*path*)

given a path of the form: [<short filename>,<number1>,<number2>...<numbern>] identify the leo node which is in that file, and, from top to bottom, is the <number1> child of the topmost node, the <number2> child of that node, and so on.

Return None if that node can not be identified that way.

node_reference(*vnode*)
Given a position p, return the name of the node.

This is called from leo.core.leoRst.

send_head()
Common code for GET and HEAD commands.

This sends the response code and MIME headers.

Return value is either a file object (which has to be copied to the outputfile by the caller unless the command was HEAD, and must be closed by the caller under all circumstances), or None, in which case the caller has nothing further to do.

split_leo_path(*path*)
Split self.path.

write_body_pane(*f*, *p*)

write_head(*f*, *headString*, *window*)

write_leo_tree(*f*, *window*, *root*)
Wriite the entire html file to f.

write_leo_windowlist()

write_node_and_subtree(*f*, *p*)

write_path(*node*, *f*)

leo.plugins.mod_http.**loop**(*timeout=5.0*, *use_poll=0*, *map=None*)
Override the loop function of asynchore. We poll only until there is not read or write request pending.

exception leo.plugins.mod_http.**noLeoNodePath**
Bases: exceptions.Exception

Raised if the path can not be converted a filename and a series of numbers. Most likely a reference to a picture.

exception leo.plugins.mod_http.**nodeNotFound**
Bases: exceptions.Exception

leo.plugins.mod_http.**node_reference**(*vnode*)
Use by the rst3 plugin.

leo.plugins.mod_http.**onFileOpen**(*tag*, *keywords*)

leo.plugins.mod_http.**plugin_wrapper**(*tag*, *keywords*)

leo.plugins.mod_http.**poll**(*timeout=0.0*)

leo.plugins.mod_http.**reconstruct_html_from_attrs**(*attrs*, *how_much_to_ignore=0*)
Given an attribute, reconstruct the html for this node.

leo.plugins.mod_http.**set_http_attribute**(*p*, *value*)

1.2.4.64 mod_leo2ascd Module

leo.plugins.mod_leo2ascd.**CodeChunk**(*text*, *width=72*)
Split a line of text into a list of chunks not longer than width.

leo.plugins.mod_leo2ascd.**CreateAscMenu**(*tag*, *keywords*)
Create the Outline to AsciiDoc menu item in the Export menu.

leo.plugins.mod_leo2ascd.**GetAscFilename**(*c*, *p*)
Checks a node for a filename directive.

leo.plugins.mod_leo2ascd.**SectionUnderline**(*h*, *level*, *v*)
Return a section underline string.

leo.plugins.mod_leo2ascd.**WriteAll**(*c*)

leo.plugins.mod_leo2ascd.**WriteAllRoots**(*c*)
Writes @root directive and/or @ascfile directive to log pane.

leo.plugins.mod_leo2ascd.**WriteNode**(*v*, *startinglevel*, *ascFile*)
Writes the contents of the node v to the ascFile.

leo.plugins.mod_leo2ascd.**WriteTreeAsAsc**(*p*, *fn*)
Writes the tree under p to the file ascFile

leo.plugins.mod_leo2ascd.**WriteTreeOfCurrentNode**(*c*)

leo.plugins.mod_leo2ascd.**init**()
Return True if the plugin has loaded successfully.

1.2.4.65 mod_read_dir_outline Module

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 répertoire en une arborescence Leo : Chaque dossier est converti en noeud dans Leo ; son nom est placé dans l'entête du noeud et chaque nom de fichier qu'il contient est listé dans son contenu.

Feedback on this plugin can be sent to:

```
Frédéric Momméja
<frederic [point] mommeja [at] laposte [point] net>
```

class leo.plugins.mod_read_dir_outline.**controller**(*c*)
Bases: object

importDir(*dir*, *compteurglobal*)
La routine récursive de lecture des fichiers

readDir(*event=None*)

leo.plugins.mod_read_dir_outline.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.mod_read_dir_outline.**onCreate**(*tag*, *keywords*)

1.2.4.66 mod_scripting Module

1.2.4.67 mod_speedups Module

Experimental speedups

Various optimizations. Use at your own risk.

If stuff breaks, disable this plugin before reporting bugs.

leo.plugins.mod_speedups.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.mod_speedups.**os_path_expanduser_cached**(*path*, *encoding=None*)

leo.plugins.mod_speedups.**os_path_finalize_cached**(*path*, ***keys*)

leo.plugins.mod_speedups.**os_path_finalize_join_cached**(**args*, ***keys*)

leo.plugins.mod_speedups.**os_path_join_speedup**(**args*, ***kw*)

leo.plugins.mod_speedups.**speedup_toUnicodeFileEncoding**(*s*, *arg=None*)

1.2.4.68 mod_tempfname Module

1.2.4.69 mod_timestamp Module

Timestamps all save operations to show when they occur.

leo.plugins.mod_timestamp.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.mod_timestamp.**timestamp**(*tag=None*, *keywords=None*)

1.2.4.70 multifile Module

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:

```
#@verbatim
```

#@multiprefix /leo #@multipath /plugins

or:

```
#@verbatim
```

#@multiprefix /leo/

#@verbatim #@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.

leo.plugins.multifile.**addMenu**(*tag*, *keywords*)

leo.plugins.multifile.**decoratedOpenFileForWriting**(*self*, *root*, *fileName*, *toString*)

leo.plugins.multifile.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.multifile.**insertDirectoryString**(*c*)

leo.plugins.multifile.**scanForMultiPath**(*c*)
Return a dictionary whose keys are fileNames and whose values are lists of paths to which the fileName is to be written. New in version 0.6 of this plugin: use ';' to separate paths in @multipath statements.

leo.plugins.multifile.**stop**(*tag*, *keywords*)

1.2.4.71 nav_qt Module

1.2.4.72 nested_splitter Module

1.2.4.73 niceNosent Module

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.

leo.plugins.niceNosent.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.niceNosent.**onPostSave**(*tag=None*, *keywords=None*)
After saving an @nosent file, replace all tabs with spaces.

leo.plugins.niceNosent.**onPreSave**(*tag=None*, *keywords=None*)
Before saving an @nosent file, make sure that all nodes have a blank line at the end.

1.2.4.74 nodeActions Module

Allows the definition of double-click actions.

The double-click-icon-box command causes 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 The double-click-icon-box command on an @file type node will save the file to disk before executing the script.

False The double-click-icon-box command 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 double-click-icon-box command matches the headline of the node against the patterns starting from the first sub-node under the "nodeActions" 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

The double-click-icon-box command 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() + '"')
```

leo.plugins.nodeActions.**applyNodeAction**(*pScript*, *pClicked*, *c*)

leo.plugins.nodeActions.**doNodeAction**(*pClicked*, *c*)

leo.plugins.nodeActions.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.nodeActions.**onIconDoubleClickNA**(*tag*, *keywords*)

leo.plugins.nodeActions.**shellScriptInWindowNA**(*c*, *script*)

1.2.4.75 nodediff Module

Provides commands to run text diffs on node bodies within Leo.

By Jacob M. Peck

Configuration Settings

This plugin is configured with the following @settings:

@string node-diff-style

One of 'compare', 'ndiff', or 'unified_diff'. Chooses which diff output method difflib uses to provide the diff. Defaults to 'compare'.

The various diff output methods are explained in the difflib documentation.

Commands

This plugin defines the following commands:

diff-marked

Runs a diff on the marked nodes. Only works if there are exactly two nodes marked in the current outline.

diff-selected

Runs a diff on the selected nodes. Only works if there are exactly two selected nodes in the current outline.

diff-subtree

Runs a diff on the children of the currently selected node. Only works if the selected node has exactly two children.

diff-saved

Runs a diff on the current node and the same node in the .leo file on disk, i.e. changes in the node since last save. **This does not work** for derived @<files> (@auto, @edit, etc.), only nodes which are part of the Leo outline itself.

diff-vcs

Runs a diff on the current node and the same node in the most recent commit of the .leo file to a VCS like git or bzr (currently only git and bzr supported), i.e. changes in the node since last commit. **This does not work** for derived @<files> (@auto, @edit, etc.), only nodes which are part of the Leo outline itself.

Common Usage

For those who don't use marked nodes for much else, the 'diff-marked' option is probably the best. Mark two nodes and then execute 'diff-marked'.

The 'diff-subtree' option is the second most common option, and makes a lot of sense for those who use clones regularly. Create an organizer node and clone your two nodes to be diffed under that organizer node, then select the organizer node and execute 'diff-subtree'.

The 'diff-selected' option is for those who use the mouse. Using Ctrl+click or Shift+click, select exactly two nodes in the outline pane, and then execute 'diff-selected'.

Scripting

nodediff.py can be used by scripts to run any of the three styles of diff. The following are available to scripts, and all of them take a list of two positions as input:

```
c.theNodeDiffController.run_compare()
c.theNodeDiffController.run_ndiff()
c.theNodeDiffController.run_unified_diff()
```

class leo.plugins.nodediff.**NodeDiffController**(*c*)

Bases: object

get_marked()

get_selection()

get_subtree()

reloadSettings()

run_appropriate_diff(*ns*)

run_compare(*l*)

run_diff_on_marked(*event=None*)

Runs a diff on the marked nodes. Will only work if exactly 2 marked nodes exist in the outline.

run_diff_on_saved(*event=None*)
run_diff_on_saved - compare current node content to saved content

Parameters

- *event*: Leo event

run_diff_on_selected(*event=None*)
Runs a diff on the selected nodes. Will only work if exactly two nodes are selected.

run_diff_on_subtree(*event=None*)
Runs a diff on the children of the currently selected node. Will only work if the node has exactly two children.

run_diff_on_vcs(*event=None*)
run_diff_on_vcs - try and check out the previous version of the Leo file and compare a node with the same gnx in that file with the current node

Parameters

- *event*: Leo event

run_ndiff(*l*)

run_unified_diff(*l*)

leo.plugins.nodediff.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.nodediff.**onCreate**(*tag*, *keys*)

1.2.4.76 nodetags Module

1.2.4.77 nodewatch Module

1.2.4.78 notebook Module

1.2.4.79 open_shell Module

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.

leo.plugins.open_shell.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.open_shell.**onCreate**(*tag*, *keywords*)

class leo.plugins.open_shell.**pluginController**(*c*)
Bases: object

launchCmd(*event=None*)

launchExplorer(*event=None*)

launchxTerm(*event=None*)

load_menu()

1.2.4.80 outline_export Module

Modifies the way exported outlines are written.

leo.plugins.outline_export.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.outline_export.**newMoreHead**(*self*, *firstLevel*, *useVerticalBar=True*)

leo.plugins.outline_export.**onStart**(*tag*, *keywords*)

1.2.4.81 paste_as_headlines Module

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.

leo.plugins.paste_as_headlines.**createPasteAsHeadlinesMenu**(*tag*, *keywords*)

leo.plugins.paste_as_headlines.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.paste_as_headlines.**paste_as_headlines**(*c*)

1.2.4.82 pluginsTest Module

1.2.4.83 plugins_menu Module

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.

class leo.plugins.plugins_menu.**PlugIn**(*plgMod*, *c=None*)
Bases: object

A class to hold information about one plugin

about(*event=None*)
Put information about this plugin in a scrolledMessage dialog.

create_menu()
Add items in the main menu for each decorated command in this plugin. The g.command decorator sets func.is_command & func.command_name.

getNiceName(*name*)
Return a nice version of the plugin name

Historically some plugins had "at_" and "mod_" prefixes to their name which makes the name look a little ugly in the lists. There is no real reason why the majority of users need to know the underlying name so here we create a nice readable version.

properties(*event=None*)
Display a modal properties dialog for this plugin

updateConfiguration(*data*)
Update the config object from the dialog 'data' structure

writeConfiguration()
Write the configuration to a file.

leo.plugins.plugins_menu.**addPluginMenuItem**(*p*, *c*)
@param p: Plugin object for one currently loaded plugin @param c: Leo-editor "commander" for the current .leo file

leo.plugins.plugins_menu.**add_menu_from_settings**(*c*)

leo.plugins.plugins_menu.**createPluginsMenu**(*tag*, *keywords*)
Create the plugins menu: calld from create-optional-menus hook.

leo.plugins.plugins_menu.**init**()
Return True if the plugin has loaded successfully.

1.2.4.84 pretty_print Module

Customizes pretty printing.

The plugin creates a do-nothing subclass of Leo's tokenizing pretty printer. To customize, simply override in this file the methods of the base prettyPrinter class in leoBeautify.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.

class leo.plugins.pretty_print.**MyPrettyPrinter**(*c*)
Bases: leo.core.leoBeautify.PythonTokenBeautifier

An example subclass of Leo's PrettyPrinter class.

Not all the base class methods are shown here: just the ones you are likely to want to override.

leo.plugins.pretty_print.**init**()
Return True if the plugin has loaded successfully.

1.2.4.85 projectwizard Module

1.2.4.86 python_terminal Module

1.2.4.87 qtGui Module

qt gui plugin.

1.2.4.88 qt_main Module

class leo.plugins.qt_main.**Ui_MainWindow**
Bases: object

retranslateUi(*MainWindow*)

setupUi(*MainWindow*)

1.2.4.89 qt_quicksearch Module

1.2.4.90 qtframecommands Module

1.2.4.91 quickMove Module

1.2.4.92 quicksearch Module

1.2.4.93 quit_leo Module

Shows how to force Leo to quit.

leo.plugins.quit_leo.**init**()
Return True if the plugin has loaded successfully.

1.2.4.94 read_only_nodes Module

1.2.4.95 redirect_to_log Module

Sends all output to the log pane.

leo.plugins.redirect_to_log.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.redirect_to_log.**onStart**(*tag*, *keywords*)

1.2.4.96 `rss` Module

Adds primitive RSS reader functionality to Leo.

By Jacob M. Peck.

RSS feeds

This plugin requires the python module 'feedparser' to be installed.

This plugin operates on RSS feed definitions, which are defined as nodes with headlines that start with *@feed*, and with bodies that contain a valid *@url* directive.

For example, the following is a valid feed definition:

```
@feed  Hack a Day
    @url http://feeds2.feedburner.com/hackaday/LgoM

    Hack a Day's feed.  Awesome tech stuff.
```

Each *@feed* node also stores a viewed history of previous stories, so that the next time the feed is parsed, you will only see new stories. This history can be reset with the *rss-clear-etc* commands below.

Important Note

This plugin currently doesn't have any undo capability - any changes performed by the following commands are not undoable.

Configuration Settings

This plugin is configured with the following @settings:

@string rss-date-format

Format string to provide datetime.time.strftime, to format entry dates. Defaults to '%Y-%m-%d %I:%M %p' if not provided.

@bool rss-sort-newest-first

If True, newest entries are placed before older entries. If False, older entries are placed before newer entries.

@string rss-headline-format

The format of an entry headline, specified with various tokens. Defaults to '[<date>] <title>' if not provided.

Valid tokens are:

<date> - the date, formatted according to *@string rss-date-format*
<title> - the entry title

<link> - the entry link (not recommended in headline)
<summary> - the entry summary (extremely not recommeded in headline)

Anything that isn't a valid token is retained untouched, such as the square brackets in the default setting.

@data rss-body-format

The body of this node will provide the structure of the body of parsed entry nodes. Empty lines should be denoted with 'n' on a line by itself. It defaults to the following, if not provided:

```
@url <link>


<title>
<date>


<summary>
```

Valid tokens are the same as for *@string rss-headline-format*. Any instance of ' ' on a line by itself is replaced with an empty line. All other strings that are not valid tokens are retained untouched, such as the *@url* directive in the default.

Commands

This plugin uses commands to operate on these *@feed* definitions. The following commands are available:

rss-parse-selected-feed

Parses the selected *@feed* node, creating entries for each story as children of the *@feed* node. Can be SLOW for large feeds.

rss-parse-all-feeds

Parses all *@feed* nodes in the current outline, creating entries for each story as children of the appropriate *@feed* nodes. Not recommended, as it can make Leo appear to be locked up while running.

rss-delete-selected-feed-stories

Deletes all the children of the selected *@feed* node.

rss-delete-all-feed-stories

Deletes all children of all *@feed* nodes in the current outline.

rss-clear-selected-feed-history

Clears the selected *@feed* node's viewed stories history.

rss-clear-all-feed-histories

Clears the viewed stories history of every *@feed* node in the current outline.

class leo.plugins.rss.**RSSController**(*c*)
Bases: object

add_entry_to_history(*feed*, *entry*)

clear_all_feed_histories(*event=None*)
Clears the viewed stories history of every *@feed* node in the current outline.

clear_history(*feed*)

clear_selected_feed_history(*event=None*)
Clears the selected *@feed* node's viewed stories history.

delete_all_feed_stories(*event=None*)
Deletes all children of all *@feed* nodes in the current outline.

delete_selected_feed_stories(*event=None*)
Deletes all the children of the selected *@feed* node.

entry_in_history(*feed*, *entry*)

get_all_feeds()

get_history(*feed*)

grab_date(*entry*)

grab_date_parsed(*entry*)

hash_entry(*entry*)

is_feed(*pos*)

parse_all_feeds(*event=None*)
Parses all *@feed* nodes in the current outline, creating entries for each story as children of the appropriate *@feed* nodes. Not recommended, as it can make Leo appear to be locked up while running.

parse_feed(*feed*)

parse_selected_feed(*event=None*)
Parses the selected *@feed* node, creating entries for each story as children of the *@feed* node. Can be SLOW for large feeds.

set_history(*feed*, *history*)

leo.plugins.rss.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.rss.**onCreate**(*tag*, *keys*)

1.2.4.97 rst3 Module

1.2.4.98 run_nodes Module

Runs a program and interface Leo through its input/output/error streams.

The double-click-icon-box command on a node whose headlines is @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.

The double-click-icon-box command on a 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.

leo.plugins.run_nodes.**CloseProcess**(*c*)

leo.plugins.run_nodes.**FindRunChildren**(*p*)

leo.plugins.run_nodes.**OnBodyKey**(*tag*, *keywords*)

leo.plugins.run_nodes.**OnIconDoubleClick**(*tag*, *keywords*)

leo.plugins.run_nodes.**OnIdle**(*tag*, *keywords*)

leo.plugins.run_nodes.**OnQuit**(*tag*, *keywords=None*)

leo.plugins.run_nodes.**OpenProcess**(*p*)

leo.plugins.run_nodes.**UpdateText**(*t*, *wcolor='black'*)

leo.plugins.run_nodes.**init**()
Return True if the plugin has loaded successfully.

class leo.plugins.run_nodes.**readingThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *verbose=None*)
Bases: threading.Thread

File = None

Text = ''

TextLock = <thread.lock object>

run()
Called automatically when the thread is created.

1.2.4.99 screen_capture Module

1.2.4.100 screencast Module

1.2.4.101 screenshots Module

1.2.4.102 script_io_to_body Module

Sends output from the Execute Script command to the end of the body pane.

leo.plugins.script_io_to_body.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.script_io_to_body.**newExecuteScript**(*self*, *event=None*, *p=None*, *script=None*, *useSelectedText=True*, *define_g=True*, *define_name='__main__'*, *silent=False*)

leo.plugins.script_io_to_body.**newPut**(*self*, *s*, **args*, ***keys*)

leo.plugins.script_io_to_body.**newPutNl**(*self*, *s*, **args*, ***keys*)

leo.plugins.script_io_to_body.**onCreate**(*tag*, *keys*)

leo.plugins.script_io_to_body.**redirect**(*c*)

leo.plugins.script_io_to_body.**undirect**(*c*)

1.2.4.103 scripts_menu Module

Creates a Scripts menu for LeoPy.leo.

leo.plugins.scripts_menu.**createScriptsMenu**(*tag*, *keywords*)

leo.plugins.scripts_menu.**init**()
Return True if the plugin has loaded successfully.

1.2.4.104 setHomeDirectory Module

Sets g.app.homeDir to a hard-coded path.

leo.plugins.setHomeDirectory.**init**()
Return True if the plugin has loaded successfully.

1.2.4.105 sftp Module

1.2.4.106 slideshow Module

Supports 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.

leo.plugins.slideshow.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.slideshow.**onCreate**(*tag*, *keys*)

class leo.plugins.slideshow.**slideshowController**(*c*)
Bases: object

createCommands()

findFirstSlideShow()

ignored(*p*)

nextSlide(*event=None*)

nextSlideShow(*event=None*)

prevSlide(*event=None*)

prevSlideShow(*event=None*)

select(*p*)
Make p the present slide, and set self.slide and maybe self.slideShowRoot.

1.2.4.107 spydershell Module

1.2.4.108 startfile Module

Launches (starts) a file given by a headline when executing the double-click-icon-box

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.

This does not work on Linux, because os.startfile does not exist.

leo.plugins.startfile.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.startfile.**onIconDoubleClick**(*tag*, *keywords*)

leo.plugins.startfile.**start_file**(*c*, *p*)

1.2.4.109 stickynotes Module

1.2.4.110 stickynotes_plus Module

1.2.4.111 swing_gui Module

1.2.4.112 systray Module

1.2.4.113 testRegisterCommand Module

A plugin to test k.registerCommand.

leo.plugins.testRegisterCommand.**hello_command**(*event*)

leo.plugins.testRegisterCommand.**hello_command2**(*event*)

leo.plugins.testRegisterCommand.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.testRegisterCommand.**onCreate**(*tag*, *keys*)

1.2.4.114 textnode Module

Supports @text nodes for reading and writing external files.

This plugin has been superceded by @edit nodes.

The @text node is for embedding text files in a leo node that won't be saved with the leo file, and won't contain any sentinel leo comments. Children of @text nodes are not saved with the derived file, though they will stay in the outline. When a outline is first loaded any @text nodes are filled with the contents of the text files on disk. To refresh the contents of an @text node, execute the double-click-icon-box command on the node.

leo.plugins.textnode.**getPath**(*c*, *p*)

leo.plugins.textnode.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.textnode.**on_icondclick**(*tag*, *keywords*)

leo.plugins.textnode.**on_open**(*tag*, *keywords*)

leo.plugins.textnode.**on_save**(*tag*, *keywords*)

leo.plugins.textnode.**readtextnode**(*c*, *p*)

leo.plugins.textnode.**savetextnode**(*c*, *p*)

1.2.4.115 threadutil Module

1.2.4.116 timestamp Module

If this plugin is enabled, the following node attributes will be managed: - str_ctime: creation time - str_mtime: time node was last modified - str_atime: time node contents were last viewed

leo.plugins.timestamp.**create_node_hook**(*tag*, *keywords*)
Hooked to <create-node> = set all 3 timestamps to now

leo.plugins.timestamp.**get_timestamp_now**()
Use standard Unix timestamps

leo.plugins.timestamp.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.timestamp.**new_hook**(*tag*, *keywords*)
Hooked to <new> event, fired when a Leo file is created, which the create_node_hook doesn't handle.

leo.plugins.timestamp.**select1_hook**(*tag*, *keywords*)
Hooked to select1, which fires when focus changes Always sets str_atime to now, sets str_mtime if node body has changed

1.2.4.117 tkGui Module

1.2.4.118 todo Module

1.2.4.119 tomboy_import Module

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.

class leo.plugins.tomboy_import.**MLStripper**
Bases: HTMLParser.HTMLParser

get_fed_data()

handle_data(*data*)

leo.plugins.tomboy_import.**capturenotes**(*c*, *pos*)

leo.plugins.tomboy_import.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.tomboy_import.**onCreate**(*tag*, *keys*)

leo.plugins.tomboy_import.**parsenote**(*cont*)

leo.plugins.tomboy_import.**pos_for_gnx**(*c*, *gnx*)

leo.plugins.tomboy_import.**strip_tags**(*cont*)

leo.plugins.tomboy_import.**tomboy_act_on_node**(*c*, *p*, *event*)

leo.plugins.tomboy_import.**tomboy_install**()

1.2.4.120 trace_gc_plugin Module

Traces changes to Leo's objects at idle time.

leo.plugins.trace_gc_plugin.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.trace_gc_plugin.**printIdleGC**(*tag*, *keywords*)

leo.plugins.trace_gc_plugin.**printIdleRefs**(*tag*, *keywords*)

1.2.4.121 trace_keys Module

Traces keystrokes in the outline and body panes.

leo.plugins.trace_keys.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.trace_keys.**onKey**(*tag*, *keywords*)

1.2.4.122 trace_tags Module

Trace most common hooks, but not key, drag or idle hooks.

leo.plugins.trace_tags.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.trace_tags.**trace_tags**(*tag*, *keywords*)

1.2.4.123 `valuespace` Module

Supports Leo scripting using per-Leo-outline namespaces.

Commands

This plugin supports the following commands:

vs-create-tree

Creates a tree whose root node is named 'valuespace' containing one child node for every entry in the namespace. The headline of each child is @@*r* <*key*>, where <*key*> is one of the keys of the namespace. The body text of the child node is the value for <*key*>.

vs-dump

Prints key/value pairs of the namespace.

vs-reset

Clears the namespace.

vs-update

Scans the entire Leo outline twice, processing @=, @*a* and @*r* nodes.

Pass 1

Pass 1 evaluates all @= and @*a* nodes in the outline as follows:

@= (assignment) nodes should have headlines of the the form:

```
@= <var>
```

Pass 1 evaluates the body text and assigns the result to <*var*>.

@*a* (anchor) nodes should have headlines of one of two forms:

```
@a
@a <var>
```

The first form evaluates the script in the **parent** node of the @*a* node. Such **bare** @a nodes serve as markers that the parent contains code to be executed.

The second form evaluates the body of the **parent** of the @*a* node and assigns the result to <*var*>.

Important: Both forms of @*a* nodes support the following **@x convention** when evaluating the parent's body text. Before evaluating the body text, pass1 scans the body text looking for @*x* lines. Such lines have two forms:

1. @*x* <*python statement*>

Pass 1 executes <*python statement*>.

2. The second form spans multiple lines of the body text:

```
@x {
python statements
@x }
```

Pass 1 executes all the python statements between the *@x {* and the *@x }*

3. Assign block of text to variable:

```
@x =<var> {
Some
Text
@x }
```

Pass 1 assigns the block of text to <var>. The type of value is SList, a special subclass of standard 'list' that makes operating with string lists convenient. Notably, you can do <var>.n to get the content as plain string.

A special case of this is the "list append" notation:

```
@x =<var>+ {
Some
Text
@x }
```

This assumes that <var> is a list, and appends the content as SList to this list. You will typically do '@x var = []' earlier in the document to make this construct work.

<var> in all constructs above can be arbitrary expression that can be on left hand side of assignment. E.g. you can use foo.bar, foo['bar'], foo().bar etc.

Pass 2

Pass 2 "renders" all *@r* nodes in the outline into body text. *@r* nodes should have the form:

```
@r <expression>
```

Pass 2 evaluates <*expression*> and places the result in the body pane.

TODO: discuss SList expressions.

Evaluating expressions

All expression are evaluated in a context that predefines Leo's *c*, *g* and *p* vars. In addition, *g.vs* is a dictionary whose keys are *c.hash()* and whose values are the namespaces for each commander. This allows communication between different namespaces, while keeping namespaces generally separate.

class leo.plugins.valuespace.**ValueSpaceController**(*c=None*, *ns=None*)

Bases: object

A class supporting per-commander evaluation spaces containing @a, @r and @= nodes.

create_tree()

The vs-create-tree command.

dump()

init_ns(*ns*)
Add 'builtin' methods to namespace

let(*var*, *val*)
Enter var into self.d with the given value. Both var and val must be strings.

let_body(*var*, *val*)

let_cl(*var*, *body*)
handle @cl node

parse_body(*p*)

render_phase()
Update p's tree (or the entire tree) as follows:

- Evaluate all @= nodes and assign them to variables
- Evaluate the body of the *parent* nodes for all @a nodes.
- Read in @vsi nodes and assign to variables

render_value(*p*, *value*)
Put the rendered value in p's body pane.

reset()
The vs-reset command.

runblock(*block*)

set_c(*c*)
reconfigure vsc for new c

Needed by ipython integration

test()

untangle(*p*)

update()
The vs-update command.

update_vs()
Evaluate @r <expr> nodes, puting the result in their body text. Output @vso nodes, based on file extension

leo.plugins.valuespace.**colorize_headlines_visitor**(*c*, *p*, *item*)
Changes @thin, @auto, @shadow to bold

leo.plugins.valuespace.**get_vs**(*c*)
deal with singleton "ipython" controller

leo.plugins.valuespace.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.valuespace.**onCreate**(*tag*, *key*)

leo.plugins.valuespace.**vs_create_tree**(*event*)
Create tree from all variables.

leo.plugins.valuespace.**vs_dump**(*event*)
Dump the valuespace for this commander.

leo.plugins.valuespace.**vs_reset**(*event*)

leo.plugins.valuespace.**vs_update**(*event*)

1.2.4.124 viewrendered Module

1.2.4.125 viewrendered2 Module

1.2.4.126 vim Module

#@@language rest

Enables two-way communication with gVim (recommended) or Vim.

Commands

vim-open-file Opens the nearest ancestor @file or @clean node in vim. Leo will update the file in the outline when you save the file in vim.

vim-open-node Opens the selected node in vim. Leo will update the node in the outline when you save the file in vim.

Installation

Set the `vim_cmd` and `vim_exe` settings as shown below.

Alternatively, you can put gvim.exe is on your PATH.

Settings

@string vim_cmd The command to execute to start gvim. Something like:

```
<path-to-gvim>/gvim --servername LEO
```

@string vim_exe The path to the gvim executable.

vim_plugin_uses_tab_feature True: Leo will put the node or file in a Vim tab card.

class leo.plugins.vim.**VimCommander**(*c*, *entire_file*)

Bases: object

A class implementing the vim plugin.

check_args()

Return True of basic checks pass.

error(*s*)

Report an error.

find_path_for_node(*p*)

Search the open-files list for a file corresponding to p.

find_root(*p*)

Return the nearest ancestor @auto or @clean node.

forget_path(*path*)

Stop handling the path: - Remove the path from the list of open-with files. - Send a command to vim telling it to close the path.

get_cursor_arg()

Compute the cursor argument for vim.

load_context_menu()
Load the contextmenu plugin.

open_file(*root*)
Open the the file in vim using c.openWith.

open_in_vim()
Open p in vim, or the entire enclosing file if entire_file is True.

should_open_old_file(*path*, *root*)
Return True if we should open the old temp file.

write_root(*root*)
Return the concatenation of all bodies in p's tree.

leo.plugins.vim.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.vim.**vim_open_file_command**(*event*)
vim.py: Open the entire file in (g)vim.

leo.plugins.vim.**vim_open_node_command**(*event*)
vim.py: open the selected node in (g)vim.

1.2.4.127 word_count Module

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.

leo.plugins.word_count.**createWordCountMenu**(*tag*, *keywords*)

leo.plugins.word_count.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.word_count.**word_count**(*c*)

1.2.4.128 wikiview Module

1.2.4.129 word_export Module

Adds the Plugins:Word Export:Export menu item to format and export the selected outline to a Word document, starting Word if necessary.

leo.plugins.word_export.**cmd_Export**(*event*)
Export the current node to Word

leo.plugins.word_export.**doPara**(*word*, *text*, *style=None*)
Write a paragraph to word

leo.plugins.word_export.**getConfiguration**()
Called when the user presses the "Apply" button on the Properties form

leo.plugins.word_export.**getWordConnection**()
Get a connection to Word

leo.plugins.word_export.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.word_export.**writeNodeAndTree**(*c*, *word*, *header_style*, *level*, *maxlevel=3*, *usesections=1*, *sectionhead=''*, *vnode=None*)

Write a node and its children to Word

1.2.4.130 wxGui Module

1.2.4.131 xemacs Module

Allows you to edit nodes in emacs/xemacs.

Provides the emacs-open-node command which passes the body text of the node to emacs.

You may edit the node in the emacs buffer and changes will appear in Leo.

leo.plugins.xemacs.**init**()

Return True if the plugin has loaded successfully.

leo.plugins.xemacs.**open_in_emacs**(*tag*, *keywords*)

leo.plugins.xemacs.**open_in_emacs_command**(*event*)

Open current node in (x)emacs

Provied by xemacs.py plugin

leo.plugins.xemacs.**open_in_emacs_helper**(*c*, *p*)

1.2.4.132 xml_edit Module

Provides commands (Alt-x) for importing and exporting XML from a Leo outline. These commands are to XML what `@auto-rst` is to reStructuredText.

`xml2leo` imports an .xml file into the node following the currently selected node. `leo2xml` exports the current subtree to an .xml file the user selects.

`xml_validate`, if executed on the top node in the Leo xml tree, reports any errors in XML generation or DTD validation, based on the DTD referenced from the XML itself. If there's no DTD it reports that as an error.

`leo2xml2leo` takes the selected Leo subtree representing an XML file, converts it to XML internally, and then creates a new Leo subtree from that XML after the original, with 'NEW ' at the start of the top node's name. This updates all the headlines, so that the convenience only previews (see below) are updated. The original can be deleted if the new subtree seems correct.

Conventions

This is a valid XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dml SYSTEM "dml.dtd">
<?xml-stylesheet href="common.css"?>
<dml xmlns='http://example.com/' xmlns:other='http://other.com/'/>
  <block type='example'>Here's <other:b>some</other:b> text</block>
</dml>
<!-- This is the last line -->
```

Note the processing instruction (xml-stylesheet), the DTD (DOCTYPE), the trailing comment (after the closing tag), and the pernicious mixed content (three separate pieces of text in the `<block/>` element). These commands attempt to deal with all of this.

- A top level Leo node is created to hold these top level parts. Its headline is the basename of the file.
- The xml declaration is placed in the body of this top level Leo node
- Below that, in the same body text, appears a simple namespace map:

```
http://example.com/
other: http://other.com/
...
```

  i.e. the default namespace first, and then any prefixed name spaces.

- Below that, in the same body text, appears the `DOCTYPE` declaration
- Children are added to this top level Leo node to represent the top level elements in the xml file. Headlines have the following meanings:
  - `? pi-target some="other" __CHK` - i.e. questionmark, space, name of processing instruction target, start of processing instruction content. Only the questionmark, which indicates the processing instruction, and the first word, which indicates the processing instruction target, matter. The remainder is just a convenience preview of the processing instruction content, which is the Leo node's body text.
  - `# This is *really* imp` - i.e. hash, space, start of comment content. Only the hash, which indicates the comment, matters. The remainder is just a convenience preview of the comment content, which is the Leo node's body text.
  - `tagname name_attribute start of element text` - i.e. the name of an element followed by a convenience preview of the element's text content. If the element has a `name` attribute that's included at the start of the text preview. Only the first word matters, it's the name of the element.
- Element's text is placed in the Leo node's body. If the element has tailing text (the `" text"` tailing the `<other:b/>` element in the above example), that occurs in the Leo node's body separated by the *tailing text sentinel*:

```
@_____TAIL_TEXT_____@
```

- Element's attributes are stored in a dict `p.v.u['_XML']['_edit']` on the Leo node. `'_XML'` is the uA prefix for these commands, and `'_edit'` is used by the `attrib_edit` plugin to identify attributes it should present to the user for editing. The `attrib_edit` plugin **should be enabled** and its `v.u mode` activated (through its submenu on the Plugins menu). The attribute edit panel initially appears as a tab in the log pane, although it can be moved around by right clicking on the pane dividers if the `viewrendered` and `free_layout` plugins are enabled.

`leo.plugins.xml_edit.`**`append_element`**(*xml_node*, *to_leo_node*)
handle appending xml_node which may be Element, Comment, or ProcessingInstruction. Recurses for Element.

`leo.plugins.xml_edit.`**`cd_here`**(*c*, *p*)
attempt to cd to the directory in effect at p according to Leo's @path concept

`leo.plugins.xml_edit.`**`get_element`**(*leo_node*)
recursively read from leo nodes and write into an Element tree

`leo.plugins.xml_edit.`**`get_tag`**(*xml_node*, *attrib=None*)
replace {http://full.name.space.com/}element with fns:element

`leo.plugins.xml_edit.`**`init`**()
Return True if the plugin has loaded successfully.

`leo.plugins.xml_edit.`**`leo2xml`**(*event*)
wrapper to write xml for current node

leo.plugins.xml_edit.**leo2xml2leo**(*event*)
wrapper to cycle leo->xml->leo, mostly to clean up headers

leo.plugins.xml_edit.**make_tag**(*tag*)
replace fns:element with {http://full.name.space.com/}element

leo.plugins.xml_edit.**xml2leo**(*event*, *from_string=None*)
handle import of an .xml file, places new subtree after c.p

leo.plugins.xml_edit.**xml_for_subtree**(*nd*)
get the xml for the subtree at nd

leo.plugins.xml_edit.**xml_validate**(*event*)
Perform DTD validation on the xml and return error output or an empty string if there is none

1.2.4.133 xsltWithNodes Module

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.

leo.plugins.xsltWithNodes.**addMenu**(*tag*, *keywords*)

leo.plugins.xsltWithNodes.**addXSLTElement**(*c*, *element*)
adds some xslt to the text node

leo.plugins.xsltWithNodes.**addXSLTNode**(*c*)
creates a node and inserts some xslt boilerplate

leo.plugins.xsltWithNodes.**cleanString**(*data*)
This method cleans a string up for the processor. It currently just removes leading and trailing whitespace

leo.plugins.xsltWithNodes.**doMinidomTest**(*c*)
This def performs a simple test on a node. Can the data be successfully parsed by minidom or not? Results are output to the log.

leo.plugins.xsltWithNodes.**getString**(*c*)
This def turns a node into a string using Leo's file-nosent write logic.

leo.plugins.xsltWithNodes.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.xsltWithNodes.**jumpToStyleNode**(*c*)
Simple method that jumps us to the current XSLT node

leo.plugins.xsltWithNodes.**processDocumentNode**(*c*)
this executes the stylesheet node against the current node

leo.plugins.xsltWithNodes.**setStyleNode**(*c*)
this command sets what the current style node is

leo.plugins.xsltWithNodes.**styleNodeSelected**(*c*)
Determines if a XSLT Style node has not been selected

1.2.4.134 zenity_file_dialogs Module

Replaces the tk 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 tk dialogs.

leo.plugins.zenity_file_dialogs.**callZenity**(*title*, *multiple=False*, *save=False*, *test=False*)

leo.plugins.zenity_file_dialogs.**init**()
Return True if the plugin has loaded successfully.

leo.plugins.zenity_file_dialogs.**onStart2**(*tag*, *keywords*)
Replace tkfile open/save method with external calls to zenity.

leo.plugins.zenity_file_dialogs.**runOpenFileDialog**(*title=None*, *filetypes=None*, *defaultextension=None*, *multiple=False*)
Call zenity's open file(s) dialog.

leo.plugins.zenity_file_dialogs.**runSaveFileDialog**(*initialfile=None*, *title=None*, *filetypes=None*, *defaultextension=None*)
Call zenity's save file dialog.

leo.plugins.zenity_file_dialogs.**testForZenity**()

1.2.4.135 Subpackages

pygeotag Package

pygeotag Module

class leo.plugins.pygeotag.pygeotag.**GeoTagRequestHandler**(*request*, *client_address*, *server*)
Bases: BaseHTTPServer.BaseHTTPRequestHandler

do_GET()

log_message()

staticMap = {'': 'template.html', 'jquery.js': 'jquery.js', 'jquery.json-2.2.min.js'

class leo.plugins.pygeotag.pygeotag.**PyGeoTag**(*callback=None*, *synchronous=False*)
Bases: object

callback(*data*)

get_position(*data={}*)

init_server()

open_server_page()

request_position(*data={}*)

show_position(*data={}*)

start_server()

stop_server()

class leo.plugins.pygeotag.pygeotag.**QueueTimeout**(*maxsize=0*)
Bases: Queue.Queue

from http://stackoverflow.com/questions/1564501/add-timeout-argument-to-pythons-queue-join

by Lukáš Lalinský

exception NotFinished
Bases: exceptions.Exception

join_with_timeout(*timeout*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z