The Essentials of Leo

Leo's window, outlines & clones

Leo organizes all data into **nodes** containing a **headline**, an expandable page of **body text** and a (possibly empty) subtree of **descendant nodes**. The **contents** of a node are its headline and body text. Leo's main window contains an **outline pane** at the top left, a **log pane** at the top right, and a **body pane** at the bottom. The outline pane shows headlines; the body pane shows the body text of the selected headline. The log pane displays messages from Leo.

Small red arrows mark **cloned nodes** (clones). Clones share the same content and descendants. All clones of the same node are equivalent, so changing the contents of one node (call it node N) changes the contents of all nodes cloned to N. Furthermore, inserting, deleting or changing any descendant of node N causes the corresponding insertion, deletion or change in all other nodes cloned to N.

Organizer nodes: relations, views and tasks

Clones allow you to organize data in multiple ways within the same outline. You do not have to choose a single 'correct' organization: you can organize your data in as many ways as you like.

You can use **organizer node**s to create explicit relations (relationships) among the data in your outline. The headline of the organizer node is the relationship's name. Now make clones of nodes from other parts of the outline that participate in the relation. Drag the newly created clones so they become children of the organizer node. You may want to create other (non-cloned) children of the organizer node that contain data found nowhere else in the outline. Voilà: you have just created the set of all nodes that pertain to the relationship. The organizer node *is* the relation; the terms relation, relationship and organizer node are interchangeable.

Two kinds of relations deserve special mention. A **view** is relation (an organizer node) containing all nodes related to a desired view (or slice) of the data in the outline. Similarly, a **task** is an organizer node containing all nodes related to a task. Relations are not limited to tasks or views: the notion of relationship is completely general.

Plugins

Plugins are Python (.py) files contained in Leo's plugins folder. Users can write plugins to change how Leo works and to add new features without changing Leo's core code. We'll discuss plugins in more detail later.

Derived files and special nodes

Leo can generate many external files called **derived files** from an outline. Leo outlines can organize files throughout your computer's file system. Nodes whose headlines start with '@' are **special nodes**. Several special nodes indicate the root of a tree that generates a derived file:

`@asis` *`filename`*	Creates a derived file. Ignores all markup in body text.
`@file` *`filename`*	Creates a derived file. Duplicates outline structure in .leo file.
`@noref` *`filename`*	Creates a derived file. Ignores all section references.
`@nosent` *`filename`*	Creates a derived file without sentinels.
`@thin` *`filename`*	(Recommended) Like @file; the derived file contains all outline structure.

Leo recognizes several other special nodes:

`@settings`	Defines settings local to the .leo file.
`@url` *`url`*	Double-clicking the node's icon opens the url in Leo or in a browser.

The scripting plugin scans for the following nodes when opening an outline:

`@button`	Creates a button in the **icon area** at the top of the Leo window.
`@plugin` *`plugin`*	Enables a plugin if the plugin has not already been enabled.
`@script`	Executes a script when opening the outline. This is a *security risk*: it is disabled by default.

Plugins, @button nodes and @script nodes can create other kinds of special nodes:

`@suite`	Creates a suite of unit tests from script in body. Created by @button.
`@test`	Creates a unit text from script in body. Created by @button.
`@rst`	Outputs a tree containing markup for reStructuredText. rst2 plugin.
`@run` *`command`*	Double clicking the node's icon executes the command.

Markup for scripts

Leo's execute script command **preprocesses** the script to be executed by scanning a node N and its descendents looking for markup. **Markup** is special syntax that controls this preprocessing. If node N contains no markup, the resulting script is just N's body text. Otherwise, the preprocessed script will include text from descendent nodes as described below. The main kinds of markup are section references, directives and doc parts.

1. **Section references** have the form:

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

The << and >> must appear on the same line. Conversely, *any* line containing << and >> is a section reference, regardless of context. However, section references are not recognized in doc parts.

Section references are functional pseudo-code: while preprocessing a script, Leo replaces section references by the actual text of the section's definition. Sections are defined in **section definition nodes**, whose headlines start with a section reference and whose body text defines the section. Each section definition node must descend from the node containing the section reference.

2. **Directives** start with '@' in the leftmost column of body text. Directives specify options and control Leo's operation.

The @others directive is the minimal markup needed to organize scripts. @others tells Leo to insert the preprocessed text of all descendent nodes (except section definition nodes) at the spot at which the @others directive occurs. Nodes are inserted in **outline order**, the order in which nodes appear in the outline. **Important**: Leo adds the whitespace preceding the @others directive to the indentation of all preprocessed text.

Using @others is more convenient than using section references. Use @others when the order of included text does not matter:

```
class myClass:
    @others # Include the methods of the class. Order doesn't matter.
```

Use section references when the order of included text *does* matter. In the following script, for example, << imports >> ensures that imports come first. The @others directive then includes the body text of all other descendent nodes.

```
@language python
<< imports>>
@others # Define classes & functions in child nodes.
main()
```

Here is a list of all of Leo's standard directives. **Important**: plugins may define other directives.

`@`*`whitespace`* (or `@doc`)	Starts a doc part & ends code part.
`@all`	Like @others, but includes all descendent nodes.
`@c, @code`	Starts a code part and ends a doc part.
`@color`	Enables syntax coloring.
`@delims`	Temporarily changes comment delims.
`@nocolor, @killcolor`	Disables syntax coloring.
`@comment`	Sets comment delimiters in external (derived) files.
`@language` *`languagename`*	Sets language for syntax coloring and comments.
`@lineending` *`lineending`*	Sets ending of lines in derived files.
`@others`	Inserts body text of all descendents except definition nodes.
`@pagewidth` *`n`*	Sets page width for justifying comments in doc parts.
`@path` *`path`*	Sets prefix to use in relative file names in @file nodes, etc.
`@root` *`filename`*	Marks the root of a tree that creates an external file.
`@raw, @end_raw`	Inhibits sections references in a range of text. (@file only.)
`@tabwidth` *`n`*	Sets width of tabs (negative widths convert tabs to spaces.)
`@wrap, @nowrap`	Controls wrapping of text in body pane.

3. **Doc parts** start with the '@' directive and continue until the end of the body text or until the '@c' directive. Body text not in a doc part is in a **code part**. Here is an example of a doc part.

```
@ This is a doc part. Doc parts may span many lines. Leo converts doc parts to comments.
Leo reformats the doc part by justifying lines so they are no longer than the page width.
@c
```

Leo reformats doc parts by justifying the text into comment lines. The @pagewidth directive controls the length of these comment lines. The @language and @comment directives specify the comment delimiters used in doc parts.

Scripting Leo

The Execute Script command preprocesses the selected text of the presently selected outline node, or the entire text of the node if there is no selected text. See the section called “Markup for Scripts” for a discussion of how Leo preprocesses scripts. That section also discusses how Leo organizes scripts using outlines. Conversely, scripts can use outlines to organize their data. To write such scripts you must understand at least the basics of Leo’s modules and classes...

Leo’s modules and classes

Leo’s source code is organized as a collection of modules. The following paragraphs describe five of the most important modules. See LeoPy.leo (Leo’s full source code) for more details: scripts have full access to *all* of Leo’s classes and data.

1. The **leoGlobals** module contains utility functions. By convention, in Leo’s code **g** is always the leoGlobals module.

2. The **leoApp** module defines a class representing the entire Leo application. **g.app** is the singleton object of this class: the **application object.** The instance variables of the application object are Leo’s global variables.

3. The **leoCommands** module defines the Commands class. A **commander** is an instance of this class. Commanders contain the operations that can be performed on a *particular* outline. Each open Leo outline has its own commander. By convention, in Leo’s code **c** is always a commander.

4. The **leoFrame** module defines the base leoFrame class for objects that create and manage the visual appearance of Leo’s windows and panes. The leoTkinterFrame and leoTkinterTree modules contain subclasses of the base classes in the leoFrame module.

A **frame** (an instance of the leoFrame class, or a subclass) contains all the internal data needed to manage a Leo window. **c.frame** is the frame associated with commander c. If f is a frame, **f.c** is the frame’s commander, **f.body** is the frame’s body pane, **f.tree** is the frame’s outline pane, and **f.log** is the frame’s log pane.

5. The **leoNodes** module defines several classes that implement Leo’s fundamental data structures. These classes are complicated. Happily, scripts can and should ignore these complications by accessing nodes the using the position class.

A **position** (an instance of the position class) is the state of some traversal of an outline. Equivalently, a position is a particular visual place in an outline. Cloned nodes may appear many times in an outline. Non-cloned nodes may also appear in many places in an outline: consider a non-cloned descendant of a cloned node. By convention, in Leo’s code p is a position.

Predefined symbols in scripts

When executing scripts Leo predefines the following three symbols: **c** is the commander of the outline in which the script is defined, **g** is the leoGlobals module and **p** is the position of the selected node in c’s outline.

Accessing data

Scripts should get data using high-level access methods to get and set data. Here are some important getters:

`g.app`	The application object. Its ivars are Leo’s global variables.
`g.app.windowList`	The list of all open frames.
`c.currentPosition()`	The position of the selected node.
`c.rootPosition()`	The position of the first node in the outline.
`p.headString()`	The headline of position p.
`p.bodyString()`	The body text of position p.
`p.childIndex()`	The number of siblings that precede p.
`p.numberOfChildren()`	The number of p’s children.
`p.level()`	The number of p’s ancestors.
`p.hasChildren()`	True if p has children.
`p.isAncestorOf(p2)`	True if p2 is a child, grandchild, etc. of p.
`p.isCloned()`	True if p is a clone.
`p.isDirty()`	True if p’s contents have been changed.
`p.isExpanded()`	True if p has children and p’s outline is expanded.
`p.isMarked()`	True if p’s headline is marked.
`p.isVisible()`	True if all of p’s ancestors are expanded.

And here are some important setters:

`p.setBodyStringOrPane(s)`	Set body text of p to s.
`p.setHeadString(s)`	Set headline of p to s.

Traversing outlines

Scripts can visit some or all of the nodes of a Leo outline using the following iterators:

`c.allNodes_iter`	All positions in the outline, in outline order.
`p.children_iter()`	All children of p.
`p.parents_iter()`	All parents of p.
`p.siblings_iter()`	All siblings of p, including p.
`p.following_siblings_iter()`	All siblings following p, not including p.

The following prints all the nodes of an outline, properly indented:

```
for p in c.allNodes_iter():
    print ' '*p.level(), p.headString()
```

Executing commands from scripts

Scripts may open other Leo outlines, and insert, delete or change any node in any open outline.

`ok,frame = g.openWithFileName(path,c)`	Opens the .leo file found at path.
`c.deleteOutline()`	Deletes the selected node.
`c.insertHeadline()`	Inserts a new node after present position.

For more examples, see Chapter 7: Scripting Leo with Python, in Leo's Users Guide.

Bringing scripts to data

The scripting plugin creates two buttons in the icon area at the top of the Leo window. The **Run Script** button executes the script in the selected node just like the Execute Script command. The **Script Button** button creates a new button whose headline is the headline of the presently selected node, call it node N. Pressing this button executes the script in node N with p predefined as c.currentPosition() *at the time the script is executed*. This clever trick brings the script to the data in the selected outline.

Unit Testing with Leo

test.leo contains all of Leo's unit tests. An @button node in test.leo creates a blue Unit Test button in the icon area. This button is an excellent example of bringing scripts to data. This button executes all the unit tests specified by @test and @suite nodes in the selected outline.

@test nodes greatly simplify unit testing. The Unit Test button creates a unit test from the body text of each @test node. In effect, the Unit Test button automatically creates an instance of unittest.TestCase whose run method is the body text of the @test node. There is no need to create TestCase objects explicitly!

When the Unit Test script button finds an @suite node it executes the script in its body text. This script should set:

```
g.app.scriptDict['suite'] = suite
```

where suite is a suite of unit tests. The Unit Test button then runs that suite of unit tests.

Plugins and hooks

Plugins are Python (.py) files in Leo's plugin subdirectory. It is easy to create new plugins: Leo's users have contributed dozens of plugins that extend Leo's capabilities in new ways. Leo imports all enabled plugins during startup. The file pluginsManager.txt lists all enabled plugins. You can use the plugin manager plugin to control plugins without updating pluginsManager.txt by hand.

Plugins can override any class, method or function in Leo's **core**, the files in Leo's src subdirectory (the files derived from LeoPy.leo). Besides altering Leo's core, plugins can register functions called **hooks** that Leo calls at various times during Leo's execution. Events that trigger hooks include key pressed events, screen drawing events, node selection events and many others. When importing a plugin, Leo will call the top-level init() function if it exists. This function should register the plugin's hooks by calling leoPlugins.registerHandler.

For full details about hooks and events see Chapter 8: Customizing Leo, in Leo's Users Guide. The file leoPlugins.leo contains all plugins that are presently distributed with Leo; studying these plugins is a good way of learning to write your own plugins.

Contacts

Leo's home page: google edreamleo *or* http://webpages.charter.net/edreamleo/front.html

Edward K. Ream: edreamleo@charter.net, 166 N. Prospect Ave., Madison WI 53726, (608) 231-0766