Grunt

Grunt is a scripting system for the Windows Shell based on the Lua programming language. In contrast to other scripting systems such as Windows Power Shell, it is based on graphical user interface concepts rather than a command line and it is compact and self-contained. It was designed originally for automating Portable Applications on USB Flash Drives and for complex "suction" backup routines on removable disk drives. Headline features:

- Fully featured scripting language based on Lua.
- System Tray icon and right-click menu customisable from Lua.
- Ability to start and manage application and command processes.
- File management facilities based on the Shell Object hierarchy.
- Ability to arrange application windows on multiple monitors.
- Support for mount and dismount of removable drives.
- Ability to spawn a secondary script with elevated privileges under Vista.
- Sophisticated time and date handling with scheduling.

Normally, executing Grunt installs a System Tray icon which may be double-clicked to show a console window. The icon has a script-defined right click menu. It is possible to suppress the installation of the tray icon using a command line switch. Whether or not the tray icon is present, the console window is shown if an error occurs in the script or if the script executes the print function.

Lua

Grunt uses the Lua 5.1 scripting language as defined in Lua Reference Manual 5.1 (http://www.lua.org/manual/ or in print from Amazon.com) and described in more detail in Programming in Lua, 2nd Edition (available in print from Amazon.com). The language has been greatly extended from this base. The following changes have been made to the standard language described in the reference documents:

1. The print command has been replaced by a reference to grunt.print described below, which offers enhanced capabilities and which outputs to the Grunt Console. Grunt.print and print are synonymous.
2. The table library is not available. The functionality of this library has been replaced by object oriented mechanisms accessed from the new library. The combination of the Table class and the List class provide all the functionality of the table library in standard Lua.

Command Line and Script Files

Grunt is implemented in a standalone file called grunt.exe as distributed. This file is relatively light-weight at less than 500KB and has no dependencies beyond the basic operating system. It does not require any installation and does not leave any footprint on machines on which it is run (except when explicitly scripted). The name of the file may be freely changed (although the extension must remain as .exe).

In the simplest case, when Grunt is executed without parameters it will load and run a script file in the same folder as the .exe and with the same name but the extension .gnt. For example, F:\grunt.exe would find and load a script file F:\grunt.gnt. An alternative file name for the Lua file can be specified on the command line when running the .exe, although this must have the extension .gnt which must be specified, for example F:\grunt.exe "myscript.gnt". Alternatively, a full path including the file name may be specified, for example F:\grunt.exe "C:\scripts\myscript.gnt".

For more complex multi-file scripting a script folder may be used to store a collection of script and other files. The folder name must have the extension .gnt and it must contain a file named init.gnt. If the script folder has the same root name as the .exe file and is located in the same folder, no command line is required. For example a renamed grunt executable C:\scripts\frogs.exe would find a script C:\scripts\frogs.gnt\init.gnt. Alternatively the folder name or the complete path to this folder may be specified as the command line. When a script folder is used, additional Lua or binary libraries may be loaded from this folder and sub-folders within it using the require statement described in section 5.3 of the Lua Reference Manual 5.1. Such libraries may have the extension .gnt or .lua for Lua code or .dll for C code.

The command line may also include switches, which must be positioned after any script file or folder that is specified. The available switches are as follows:

/noicon – This suppresses the installation of the tray icon on start-up. The icon can still be installed from the script using grunt.seticon.

It is recommended to install Lua For Windows (http://luaforge.net/projects/luaforwindows/)on the computer used for developing Grunt scripts as this offers an excellent syntax highlighting Lua code editor.

The runtime distribution for Grunt includes an examples directory with many example Lua scripts and these should be consulted in conjunction with this manual which has few examples.

The Grunt Console

The Grunt console is displayed if an executing script encounters an error, if the print function is executed by a script or if the user double-clicks the Grunt taskbar icon. Its main purpose is to provide a surface for display of error, diagnostic and information messages. It also provides several useful menu options such as the ability to re-start and debug the script and the ability to terminate the Grunt program if the script does not include that option.

Scripts can also add menu commands to the console menu to provide additional script management facilities.

Concepts

Lua Specifics

Lua is a relatively conventional structured procedural scripting language which is very well documented in the references already mentioned. It has some unusual features however and a brief introduction to these will assist in understanding the Grunt extensions.

Functions in Lua can return more than one value which you assign using list syntax:

```
v1, v2, v3 = afunction()
```

You do not have to assign all the return values – any that are not assigned will be discarded. If a function is used within an expression, only the first return value is used.

Lua uses an escape character system in strings which is similar to that used in the C language and which means that the backslash character is interpreted specially giving a problem with the backslash path separator in Windows. However Lua also provides an alternative string delimiter which does not use escaping:

```
[[“C:\test\fish.exe” /c]]
```

This gives a string as printed including the quote marks but excluding the double-square brackets which are the replacement delimiters. You can even define multiline strings literally as all characters within the brackets, including newline characters and tabs are taken literally. However an exception is that if the very first character is a newline character, it is dropped.

Notation

When a function or method is introduced in this document, optional parameters are shown as syntax in square brackets:

function(p1 [, p2])

P1 is required while p2 is optional, so assuming these are string parameters, the call could be made in either of the following forms:

```
function(“test”, “this”)

function(“test”)
```

Notice the difference between the following two notations:

function([p1] [, p2])

function([p1 [, p2]])

In the first case, you can supply p1 or p2 or both whereas in the second case only p1 or both parameters is valid. In the first case, the parameters are distinguished by type (or sometimes content) while in the second they are distinguished purely by position. However in both cases, any parameters that are provided must be in the correct relative order.

Return parameters are always positional and optional. Even if there is only one return parameter, you are free to ignore it by not providing a variable to receive it. It is common practice in Lua to use a variable name "_" (single underscore character) as a dummy variable to hold place for unwanted returns. If a function returns two parameters, the following calls are all valid:

```
function()

a = function()

a, b = function()

_, b = function()
```

Some functions and methods take or return a variable number of parameters and this is indicated by a triple-period in the parameter or returns list:

```
... = function( ... )
```

A few functions and methods use named parameters in "name=parameter" form. This allows for a large number of parameters most of which are optional. Lua allows function calls of the form:

```
function{ ... }
```

Which is a syntactic shortcut (the Lua documentation calls idioms like this "syntactic sugar") for:

```
function({ ... })
```

So the function is actually receiving a single anonymous table variable which is constructed in place. This table must have string keys (the parameter names) and corresponding values of any type:

```
function{param1="fred", param20=34.5}
```

Libraries

Libraries provide a namespace for related functions, menus and events. Like practically everything in Lua, libraries are implemented as tables. Some libraries are implemented by Grunt and the features they provide can be used in scripts. To use a library function, menu or event, you prefix the identifier with the library name and a period character, for example:

```
grunt.print("Hello from Grunt")
```

Classes and Objects

An object is a kind of variable that encapsulates state (data) and methods (functions which act on the state). The methods are contained in the object's class (which is shared by all objects of that class) while the state is contained in the object itself and may have different values in each object. Grunt objects can also have events. Methods are very similar to functions in libraries, except that a colon is used to separate the object name from the method name. A hidden extra first parameter is supplied by the interpreter which is a reference to the object, extending context into the method. For example, given an object drv of class Drive, a valid method call is:

```
drvtyp = drv:getinfo()
```

Grunt adds a comprehensive, general purpose system to Lua for the creation of new classes and objects and which enables Lua tables to be used in an object oriented manner. There are also a number of pre-defined classes which can be used to create specialised objects to interact with the Windows Shell. Generically, objects are created using the new library which has a specific function for the creation of objects of each class:

```
obj = new.Classname(...)
```

Some classes require or allow additional parameters to be passed to initialize the object.

Some library functions and object methods also create and return objects of specific classes.

Events

Libraries and classes may also implement events. Events allow you to define Lua functions that will be called when specific conditions occur in the Windows operating system. Events are actually fields which reference Delegate_List objects which are described in full later. For simple use, we merely need to use the add method to add a function to the appropriate field. The Event fields all begin with On so, for example, to have code execute when the battery becomes low we use the OnBatteryLow event as follows:

```
shell.OnBatteryLow:add(function() print("Battery Low") end)
```

Of course, we could define a named function and then just add the function name. The add method can be used repeatedly to add multiple functions to the same Event.

Icons

The grunt executable automatically loads all icons found in its file resources into the table grunt.icons. If there is a string resource with the same resource number, that text will be used as the index in the grunt.icons table. If there is no string resource, the icon is loaded under an integer key (which is not the same as the resource number). Icon resources can be added, changed or deleted using third party resource editing applications with no code changes required. It is also possible to build libraries of icons which can be loaded using the Lua require statement. Such libraries can be created by copying and renaming the file "temp-lib.dll" provided with the Grunt distribution, and adding icon and string resources to this file.

Note that the value in the grunt.icons table is a function (actually a closure), but this is an implementation detail – the function is called and used in C code within Grunt functions. A value from this table is simply passed to a function or method which requires an icon.

Grunt Library

The grunt library contains functions and fields connected with the operation of the Grunt scripting system itself.

Functions

grunt.setname(name)

Sets the text shown in the tooltip on the taskbar icon and in the header of dialog boxes. Until or unless this function is called, the name will be the name of the .exe file which is "Grunt" as distributed, but may be different if this file is renamed.

grunt.seticon(icon)

Sets the icon shown in the taskbar. The icon parameter can be a value from the grunt.icons table, for example:

```
grunt.seticon(grunt.icons["Lock"])
```

Or it can be Boolean true or false. False disables the tray icon while true enables it using the default icon. This can be used to remove the tray icon if it is not required, or to install it if it is disabled on startup (by command line switch or by starting through grunt.spawn). Setting a specific icon from the grunt.icons table will also enable the tray icon if it is disabled.

grunt.print(...)

Takes any number of arguments and prints them on the Grunt console, separated by spaces. This also replaces the Global print function and so may be used without the grunt library prefix. Strings are printed directly and nil is printed 'NIL'. Other types are printed as a type name, a colon and, where possible, a string representation of the value.

grunt.close([unconditional])

Shuts down the scripting system. If unconditional is supplied and evaluates to logical true, shutdown occurs unconditionally, otherwise shutdown only occurs if the console is hidden. The conditional option is useful for catching errors in scripts because the user can view the messages in the Grunt Console and then manually close the script system using the Grunt Console menu.

R1, R2, R3, R4 = grunt.locale()

Returns information about the execution environment of the scripting system as follows:

R1: A string representing the DNS name of the computer on which the script is running.

R2: A string representing the username of the account in which the script is running.

R3: A string representing the network path or drive letter containing the initial script.

R4: A string representing the directory path containing the initial script.

env = grunt.environment()

Returns a Table object which has entries with string keys and values corresponding to the environment variables of the current process.

res = grunt.spawn(scriptfile [, flags])

Creates a new Grunt process, optionally with administrator privileges, and runs the specified scriptfile in it. Scriptfile may be a full file name and path or just a name. If there is no path, the file is sought in the same folder as the original script. If the name has no extension, the extension ".gnt" is added. Flags must be a list of zero or more of the following string keys, in any order:

admin - Specifies that the new instance of Grunt should be "run as administrator" in Vista.

icon – Specifies that a tray icon should be installed for the new process.

Menus

Note: List objects and Command objects are described later in this document.

Menus in Grunt are defined by assigning a List object or a Lua table with list restrictions to a predefined field in the grunt library. The following fields are supported:

grunt.TaskMenu = { ... }

The table defines the menu that appears when the Grunt task bar icon is right-clicked.

grunt.ToolsMenu = { ... }

The table defines the Tools menu on the Grunt console.

To define a simple static menu it is easiest just to assign a table initializer directly to the appropriate field. If the menu may need to be altered programmatically it will be easiest to define it as a List object and assign this to the field. It is even possible to create several alternative menu List objects and swap them programmatically. Grunt generates menu items from the menu list each time the menu is shown.

Each item in the menu list must be one of the following:

1. A Command Object (either stateless or Boolean).
2. A one character String "-" which inserts a horizontal separator line across the menu.
3. A one character String "=" which marks the next item in the list as the default item.
4. A one character String "|" which starts a new vertical column in the menu.
5. Any String not covered above which creates a passive title in the menu.
6. A List with exactly two entries. This defines a sub-menu. Index 1 must be a String and this becomes the title of the sub-menu. Index 2 must be a menu List according to this definition. Sub-menus may be nested to any depth.

Actions are linked to menu commands by adding a notifier function to the Command object. When the menu item is clicked, the notifier function is called. Boolean Command objects appear with a check mark in the menu and clicking the item will toggle this on and off. The state is retained in the

Command object and may be interrogated when needed, or a notifier function may be attached if instant action is required.

Command objects may be disabled and enabled at any time. Disabled commands appear in menus but are greyed out and cannot be clicked.

The architecture allows for the same Command object to be linked to more than one UI element. For example the same command could appear on both the menus supported by Grunt, or on a menu and one of that menu's sub-menus. The state is synchronised between all the elements and may be changed by any of them.

Events

There is a corresponding event for each menu:

grunt.OnToolsMenu

Functions added to this Event are called immediately before showing the Tools menu. Any required alterations to the menu list or commands can be made here.

grunt.OnTaskMenu

Functions added to this Event are called immediately before showing the Task icon menu. Any required alterations to the menu list or commands can be made here.

Shell Library

The shell library contains functions which interface with the Widows Shell. See also the ShellObject class later in this document.

Functions

key = shell.messagebox(message [, icon [,buttons [,defbutton]]])

Displays a Windows message box and returns the name of the key which the user presses. Message parameter must be a string which specifies the message text to be displayed. Icon parameter is a string key which must be “none”, “stop”, “question”, “exclamation” or “asterisk”. Buttons parameter specifies the buttons to show and must be one of the string keys “ok”, “ok+cancel”, “abort+retry+ignore”, “yes+no+cancel”, “yes+no”, “retry+cancel” or “cancel+tryagain+continue”. Defbutton is a number which specifies which button is the default, numbered from 1 being the first button in the buttons string key.

shell.balloon([title [,text [,icon [,timeout]]]])

Shows, or if called without parameters, hides a balloon message on the taskbar icon. Title is a short string which appears in the heading of the balloon. Text is a longer string which appears in the body of the balloon. Icon is a string key which must be “none”, “info”, “exclamation”, “stop” or “grunt” with the last being the same as the task tray icon. Timeout is a number representing the length of time in milliseconds for which the balloon should be shown.

list = shell.monitors([full])

Returns a List object (see later) containing a Frame object (see later) for each display monitor making up the computer’s virtual screen. Full, if true, requests the full display area of each monitor; if absent or false, only the working area (less taskbars etc.) is included. The first entry in the list always represents the primary monitor.

frame = shell.screen()

Returns a Frame object (see later) representing the enclosing bounds of the entire virtual screen.

onAC, charge, capacity = shell.powerstatus()

Returns the following three pieces of information about the computer power:

onAC: Boolean true if the computer is receiving AC power.

charge: Number or nil, Battery charge status 0 .. 100 (percent) or nil if there is no battery.

capacity: Number or nil, Estimated endurance of the computer in seconds from 100 percent charge, nil if no battery.

res = shell.lockworkstation()

Locks the workstation if possible and returns Boolean true if it was successfully locked.

Events

The shell library provides the following events:

shell.OnDisplayChanged

Functions added to this Event will be called whenever the display settings change in any way. shell.monitors and shell.screen should be called again as screen or monitor coordinates may have been changed.

shell.OnSessionEnding

Functions added to this Event will be called before the user is logged out. This occurs when the computer is shutting down; restarting or the user is changing.

shell.OnBatteryLow

Functions added to this Event will be called when the available battery capacity drops below a preset threshold (usually about 10%).

shell.OnSuspending

Functions added to this Event will be called when the computer is about to be placed in power management suspension.

shell.OnResuming

Functions added to this Event will be called when the computer has just exited from power management suspension.

shell.OnThemeChanged

Functions added to this Event will be called when the desktop theme has changed.

shell.OnTimeChanged

Functions added to this Event will be called when the computer time experiences a discontinuity (for example, a time correction or daylight savings change).

shell.OnUserChanged

Functions added to this Event will be called when the user account changes.

Frame Class

The Frame class of objects represent a rectangle on the screen or a point on the screen. The Frame object also has a __tostring method allowing its value to be displayed by the print function.

Creation Function

obj = new.Frame([x , y [,w, h]])

Returns a new Frame object. If called without parameters, represents point (0, 0). If called with two parameters, represents a point. If called with four parameters, represents a rectangle.

Methods

L = obj.left(); t = obj.top(); w = obj.width(); h = obj.height()

Returns the x-coordinate of the left side of the rectangle, the y-coordinate of the top of the rectangle, the width of the rectangle or the height of the rectangle.

r = obj:right(); b = obj:bottom()

Calculates and returns the x coordinate of the right hand side of the rectangle or the y coordinate of the bottom of the rectangle.

Command Class

The command class of objects provides linkage between UI elements and actions defined as Lua functions. Commands may be stateless (for example 'cut') or may have state of any type (for example, a Boolean command to represent a switch or a String to represent a username). In the current version, command objects are used to define menu items and may be stateless or Boolean (the latter shows a toggling tick mark in the menu).

Creation Function

cmd = new.Command(text, func)

cmd = new.Command(text, init)

The first form creates a stateless command and attaches the function which will be invoked when the UI element is clicked (see add method for details of this function). The second form creates a command with state and initial value defined by the second parameter (in the current version, this must be a Boolean). Text is a string which labels the command on UI elements (for example, it becomes the menu text).

Methods

cmd:add(func [, ...])

Adds one or more notifier functions to the command. These functions will be called in arbitrary order when the command state is changed or when a stateless command is triggered. Notifier functions are called (by the system) as follows:

notifier_func(command, type [,oldstate])

Command is a reference to the Command that triggered the notifier. Type is Number 1 (other type codes are reserved for planned enhancements). Oldstate is the state of the Command prior to the change and is only present for Commands with state.

cmd:remove(func [, ...])

Removes one or more notifier functions. To use this facility, you must store the function in a variable independently of the Command so a reference to the exact same function can be given to the remove method.

type = cmd:type()

Returns a string representing the type of the Command. This will be a Lua type name or "stateless".

state = cmd:state()

Returns the current state of a command with state (or nil for a stateless command).

prev = cmd:disable([new])

If the parameter is missing or Boolean true, the Command is disabled. If it is false the command is enabled. Returns true if the Command was previously disabled, else false.

Operations

cmd([state])

Function call semantics can be used to invoke the command programmatically. This has no effect if the command is disabled. For commands with state, a state of the same type must be supplied and this is assigned provided the command is not disabled. The notifiers are only called if the state is actually changed. For a stateless command, the notifiers are always called unless the command is disabled.

Time Class

Objects of Time class fall into three distinct variations, representing an absolute time-and-date, a duration (from milliseconds up to years) or a time-of-day. The internal representation is a 64-bit floating point number which encodes days and fractions of a day. Time objects can also be used to schedule functions for execution in the future.

The absolute form represents a positive or negative offset from a datum and is stored in UTC form (GMT time zone), but the access methods convert the number to the local time zone and daylight savings offset. A resolution marker is also stored allowing the object to represent a full minute, hour, day, month or year (without the marker, millisecond resolution is stored). Most frequently, this will be used to store a date without time (day resolution).

The duration form represents a time duration, for example the difference between two absolute Time objects.

The time-of-day form is very similar to the duration form but is limited to just under one day. It is used to represent a time without a date. Mathematical operations use “clock arithmetic” so, for example, adding a duration of two hours to 23:00 gives 01:00.

Creation Function

tim = new.Time()

Creates an absolute Time object representing the current time and date.

tim = new.Time(time)

Copies a Time object.

tim = new.Time{ ... }

The Lua named parameter form is used (shortcut to passing a single table) for all other cases. The following keys are available (with example values):

year=2008 - Four-digit year (If present an absolute Time is created).

month=5 - Month as a number, 1 .. 12 (If present an absolute Time is created).

day=23 - Day of month as a number, 1 .. 31 (If present an absolute Time is created).

hour=14 - Hour of day, 0 .. 23 (If present an absolute Time or time-of-day is created).

minute=12 - Minute of hour, 0 .. 59 (If present an absolute Time or time-of-day is created).

second=5.6 - Second and fraction (If present an absolute Time or time-of-day is created).

If any of year, month or day is supplied, the Time will be absolute and the resolution is set to the smallest component supplied. Any higher components that are missing will be set to the current value. If any of hour, minute or second is supplied (and not year, month or day), a time-of-day object is created. Again, missing components larger than the smallest are filled in from current values and smaller ones are set to zero.

zoneoffset=-60 - Offset in minutes to be added to supplied time to give UTC.

Usually this can be omitted and current time zone information is used for the conversion. If supplied, this value should include any daylight saving offset applicable. It can be set to zero to indicate that the other values already represent UTC.

weeks=1.0 - Duration in weeks (and fractions of a week)

days=3.5 - Duration in days (and fractions of a day)

hours=2.8 - Duration in hours (and fractions of an hour)

minutes=56.9 - Duration in minutes (and fractions of a minute)

seconds=34.78 - Duration in seconds (and fractions of a second)

If none of the absolute/time-of-day parameters are provided a duration Time object is produced. If none of the above plural parameters are provided, the duration will be zero. All the parameters are floating point numbers (or integers), unlimited in magnitude and may be negative. The values are simply multiplied by the appropriate multiple and then summed to give the final duration.

Time Zones

As described before, both absolute Time objects and time-of-day Time objects store their value internally as UTC time (which is roughly sun time at the zero longitude meridian running through Greenwhich, England). By default, when a new Time object is created or when a Time object is accessed, local time zone information is used to convert the time back to local time (taking account both of the local time zone offset and any local daylight savings policy). However the creation and access functions all take an optional time zone offset parameter which can be used to specify a different local time or can be set to zero to specify UTC. This offset is in minutes and is added to local time to produce UTC, or subtracted from UTC to produce local time. It must include any daylight savings offset.

This architecture allows direct calculations on times from different time zones. For example if time A is created under British Summer Time (offset +60) and time B under Eastern Standard Time (offset –300), subtracting A from B will create a duration object representing the correct difference between the two times. If you create a Time whilst the machine is set to one time zone and then change it to another zone, the time will be correctly converted to the new zone on access.

However there are some limitations that need to be taken into account.

Firstly, the system only knows about the current daylight savings policy. If the policy is changed, dates before the policy change may not be converted properly. If in doubt, enter the offset explicitly when entering a date. However, the current date should always be converted properly and the changeover times are stored in a day of the week sensitive format (i.e. if the policy is ‘last Sunday in October at 02:00’ it does not need to be changed every year to reflect the actual date).

Another subtlety is the representation of resolution limited absolute Time objects. These represent the start of the period in the time zone in which they were originally entered. For example, 1st May

2005 entered in EST retains the zone offset when converted to UTC. This allows comparisons to work correctly within a time zone, but may lead to unexpected results when comparing resolution limited absolute Time objects created in different zones.

Methods

offset, name, dstoffset = tim:zone()

Returns time zone information for the current time zone of the machine. Tim must be an absolute Time object and the return values include DST information appropriate for this time and date. Offset is a Number (integer) representing the total offset from UTC in minutes comprising the zone offset and the DTS offset. Name is a String name for the time zone. Dstoffset is a Number (integer) representing the daylight savings time offset applicable (it will be zero if DST is not in effect at the represented date and time).

type, res = tim:type()

Returns the type and resolution (if applicable) of a Time object. Type is a String, “absolute”, “day” or “duration”. Res is returned only if type is absolute and is a String, “”, “year”, “month”, “day”, “hour” or “minute” (empty string means full resolution).

tim:format([fmt] [,tzo])

Returns a string representing the Time. Fmt is a string format template including substitution keys. If missing, a sensible default depending on type, resolution and magnitude is used. Tzo is the time zone offset in minutes (ignored for Duration Time objects). This value is subtracted from the stored UTC value to generate local time (it must include any daylight savings offset). If it is not supplied, the current time zone and daylight savings information is used for the conversion. Pass a value of 0 to return the string in UTC (GMT).

The following substitution keys are available for Absolute and Time-of-Day objects:

%a - Abbreviated weekday name

%A - Full weekday name

%b - Abbreviated month name

%B - Full month name

%c, %#c - Date and time representation appropriate for locale (short or long form)

%d - Day of month as decimal number (01 – 31)

%H - Hour in 24-hour format (00 – 23)

%I - Hour in 12-hour format (01 – 12)

%j - Day of year as decimal number (001 – 366)

%m - Month as decimal number (01 – 12)

%M - Minute as decimal number (00 – 59)

%p - Current locale's A.M./P.M. indicator for 12-hour clock

%S - Second as decimal number (00 – 59)

%U - Week of year as decimal number, with Sunday as first day of week (00 – 53)

%w - Weekday as decimal number (0 – 6; Sunday is 0)

%W - Week of year as decimal number, with Monday as first day of week (00 – 53)

%x, %#x - Date representation for current locale (short or long form)

%X - Time representation for current locale

%y - Year without century, as decimal number (00 – 99)

%Y - Year with century, as decimal number

%% - Percent sign

Add # after the % to remove the leading zero from any two-digit format.

The following substitution keys are available for Duration objects:

%w - Number of weeks.

%d - Number of days.

%h - Number of hours.

%m - Number of minutes.

%s - Number of seconds.

Each of the above substitutes an integer (rounded down). The same letters capitalised substitute a floating point number.

hour, minute, second = tim:time([tzo])

Returns the hour, minute and second components of an absolute or time-of-day Time. Hour and minute are integers 0 .. 23 or 0 .. 59, second is a real number 0 .. 59.999. If tzo is omitted, local time is returned, if it is 0, UTC is returned, otherwise it is the time zone offset in minutes.

year, month, day = tim:date([tzo])

Returns the year, month and day components of an absolute Time. Year is an integer (four digits), month is an integer 1 .. 12, day is an integer 1 .. 31. If tzo is omitted, local time is returned, if it is 0, UTC is returned, otherwise it is the time zone offset in minutes.

day = tim:day([tzo])

Returns the day of week represented by an absolute Time. The return is an integer 0 .. 6 where 0 is Sunday. If tzo is omitted, local time is returned, if it is 0, UTC is returned, otherwise it is the time zone offset in minutes.

val = tim:value()

Returns the value of the Time object as a real number of days. This is most useful for Duration type objects, but it works for all types.

tim:alarm([func [, repeat]])

Schedules func (a Lua function) to run at the time represented by tim. If tim is 'absolute' type, the function will run once at or shortly after that time (and the repeat value is ignored). Repeat is a number which defaults to 1 if absent and which represents the number of times the function should be executed. A repeat value of less than 1, or of Boolean true, means repeat indefinitely. For 'day' type tim, the function will be executed on or shortly after that time each day until the repeat count expires. For 'duration' type tim, Grunt will wait for that duration of time before the first execution and between subsequent executions of the function.

Only one alarm is allowed per Time object and any subsequent execution of the alarm method will overwrite any existing alarm. The alarm may be cancelled by calling the alarm method with no parameters.

Operations

Mathmatical operators (+, -, *, /)

Where A is an absolute Time object, D is a duration Time object, T is a time-of-day Time object and N is a number, the following operations are supported:

A = A + D; A = D + A; D = D + D; T = T + D; T = D + T; A = A – D; D = D – D; D = A – A; T = T – D; D = - D

D = D * N; D = D / N; N = D / D

Absolute objects with month or year resolution do not support addition or subtraction of durations (because months and years do not have fixed length). For other resolutions, the resolution remains the same and the duration is rounded down to that resolution (for example, adding 10 minutes to an object of day resolution has no effect). All operations on absolute objects are carried out on the start time only, for example D = A – A gives the duration between the start times of these objects regardless of resolution, not the "free time" between them.

Time-of-day objects use "clock arithmetic", for example, subtracting two hours 10 minutes from 01:00 gives 22:50.

Relational operators (==, ~=, <, <=, >, >=)

The relational operators are all supported as expected for comparisons between the same variant of Time object. Comparing different variants gives false except for ~= which gives true.

For limited resolution absolute Time objects, comparisons take account of resolution. For example, if A1 has a resolution of Day and A2 has full resolution, A1 == A2 and A2 == A1 are both true if A2 represents any time during the day represented by A1. In general, absolute objects are equal if the shorter resolution object is contained within (or is identical to) the longer.

Similarly, for limited resolution absolute Time objects, A1 > A2 is true only if A1 represents a time starting after the end of A2. A1 >= A2 is true if the start of A1 is after or the same as the start of A2. A1 < A2 is true if the end of A1 occurs before the start of A2 while A1 <= A2 it true if the end of A1 occurs before or on the end of A2.

Application Class

Objects of the Application class represent applications (such as notepad.exe) or command lines (such as format.exe). The object has a __tostring metamethod which prints the window title if the application is running, or the command field.

Creation Function

obj = new.Application(command [, directory [, environment]])

Returns a new Application object. Command is a string which is the command line required to start the application including any parameters (similar to the Target field in a shortcut). Directory is an optional string which sets the default directory (similar to the Start in field in a shortcut). Tip: in Lua you can use paired square brackets to delimit a string literally without escape characters, including backslashes and quote marks without requiring special handling. For example: [[“C:\Program Files\appco\theapp.exe” /c]]. Environment is an optional Table which is used to pass environment strings to the new process. The keys in this table become the names of environment variables and the values the values of the variables. If you do not supply a table, the process inherits the environment of the current process. To supply a modified current environment use grunt.environment() to get the current environment, modify the returned table and then supply that in new.Application.

Methods

res = obj:open(...)

Executes the command line storing the resulting process id and main window handle (if any). Any number of string parameters may be supplied which form a set – any combination of the allowed keys may be provided in any order. The allowed string keys are:

sync - Normally, open returns as soon as the application has started and termination is signalled asynchronously via the OnClose event, but if the sync key is present, open does not return until the application terminates.

noconsole - Applies only to console (as opposed to UI) applications and it causes the console window to be hidden.

Res returns the process exit code (Number type) for the synchronous case or a Boolean true for the asynchronous case. If the application cannot be started, returns Boolean false.

obj:close([force])

If force is absent or false, attempts to close the application by sending its top level window a WM_CLOSE message. If this fails, or if force is true, the process is terminated by force.

obj:setstate([state])

Sets the state of the top level window to one of the string keys “normal”, “minimized”, “maximized” or “hidden”. If the application is already open, its state is changed immediately. If the application is not open, the state is stored and used as the initial state when the open method is executed.

obj:setframe([frame])

Sets the position and size of the top level window. If the application is open and in the “normal” state, it is moved and optionally resized to conform to the Frame. If the application is not open, the Frame is stored and applied when the open method is executed.

title, state, frame = obj:windowstate()

If the application is running and a main window can be located for it, returns the title of the main window (string), the state of the window (string: “normal”, “minimized”, “maximized” or “hidden”), and if the state is “normal” the size and position of the window (Frame object). If the application is not running, or the main window cannot be found, returns nil.

code = obj:exitcode()

If the application has been run and has exited, returns the exit code from the process (number 0..255). If it has not yet been run, or is still running, returns nil.

Events

obj.OnClose

Functions added to this event will be called when the application process terminates.

Drive Class

Objects of the Drive class each represent one of the drive letters on the computer.

Creation Function

obj = new.Drive([name] [,free])

Returns a new Drive object. This can be used to access information about an existing drive letter, to find the drive letter containing a volume with a specified name or to locate a free drive letter on which to mount a volume. Name is a string which is either a volume label or a drive letter followed by a colon ("A:"). Free is a Boolean and if true, specifies that a free drive letter should be located. Without any parameters, the first free drive letter 'C' or above is selected ('B' then 'A' will be selected if they are free and there are no other free letters). With only the drive letter parameter, that drive letter is selected whether free or not. With a second parameter which evaluates to Boolean true, the specified drive letter is selected if it is free, otherwise the nearest free drive letter is selected. With a volume label parameter, all available drive letters are checked for a volume with that name and the drive letter of the first match is selected. The function will return nil rather than an object if a drive cannot be selected as specified.

Methods

root, letter, vol = obj:path()

Returns the drive letter represented by the object in three different forms (assuming the letter is 'X'):

root: String - Root directory form ("X:\").

letter: String – Just the drive letter (single upper-case letter) ("X").

vol: String - Volume specifier form ("\\.\X:").

type, volname, fsname, serial = obj:info()

Returns the type of drive and information about the volume (if any) in the drive. Returns nil if the drive is free or just the first parameter if the drive exists but does not have a volume (for example, a CD drive with no CD).

type: String - "fixed", "removable", "network", "optical" or "ramdisk".

volname: String – the volume name.

fsname: String – the file system name, for example "FAT", "NTFS" or "CDFS".

serial: Number – the volume serial number (a unique identifier fixed when a volume is formatted).

free, total, size = obj:space()

Returns information about the space available on a volume in the drive (returns nil if there is no volume):

free: Number – The free space in kilobytes available for writing by the current user on the volume.

total: Number – The total space in kilobytes, free and used, accessible to the current user on the volume.

size: Number – The total size in kilobytes of the volume regardless of any user quotas.

res = obj:ejectvolume([fixed] [, timeout])

Dismounts and, if supported, ejects the volume in the drive. Fixed is a Boolean parameter which, if true, allows an attempt to be made to dismount a drive even though it is fixed. This may be necessary for some e-sata and caddied drives. Timeout is the amount of time in milliseconds to wait for a volume lock (default is 10000, 10 seconds). The method returns a numeric result code. 0 means total success, less than 0 means the volume was dismounted but could not be ejected, greater than 0 means that the volume could not be dismounted (usually because it is in use).

Note: If this method fails on Windows Vista with res = 2, the reason will usually be that administrator privileges are required. One way of achieving this is with the grunt.spawn function with the “admin” flag . The other common failure code is 3 which means that some other process has a file open on the drive.

Events

obj.OnMount

Functions added to this event get called whenever a volume is mounted in the drive (for example, a DVD is inserted or a USB drive is plugged in).

obj.OnDismount

Functions added to this event get called whenever a volume is removed from the drive.

ShellObject Class

Objects of the ShellObject class represent a Windows Shell Namespace Object. The Shell Namespace is the hierarchy displayed in Windows Explorer and it comprises both the file system and also virtual objects such as printers and network locations.

Creation Function

obj = new.ShellObject([spec])

Returns a new ShellObject. Creating a new ShellObject gets an entry point into the hierarchy and you can use this object to create parents and children from this point. With no parameters, the ShellObject represents the Desktop. A string parameter is interpreted as a directory path. A numeric parameter is interpreted as a CSIDL code. These codes may be looked up in the CSIDL table. For example, to get a ShellObject for My Documents use:

```
obj = new.ShellObject(CSIDL.personal)
```

This function may return a nil object if a directory does not parse or an invalid CSIDL code is specified.

Methods

par = obj:parent([levels])

Returns a ShellObject representing the parent of the current object. Levels is a number which defaults to 1 but may be incremented to specify the grandparent, great-grandparent etc. in a single call. If you attempt to go back too far, the return is nil.

Note: See “operations” below for information on navigating to children.

sel = obj:browse([prompt [, files [, nocreatefolder]]])

Presents a shell browser dialog for the ShellObject. If the user selects a new object this is returned as a new ShellObject, otherwise returns nil. Prompt is a string which is displayed to the user (defaults to “Choose a folder”). Files is a Boolean which, if true, allows the selection of non-folders (usually files). Nocreatefolder is a Boolean which, if true, disables the “new folder” button on the dialog preventing the user from creating a new folder.

attset, mod, create, access, size = obj:attributes()

Returns a Set object as characterising the ShellObject. Set objects are described in full later, but briefly they are tables in which particular keys are either present or not. The possible keys in this case are “cancopy”, “candelete”, “canmove”, “canrename”, “compressed”, “encrypted”, “filesystem”, “folder”, “hidden”, “link”, “readonly”, “removable”, “share”, “storage” and “stream”. You can test for the presence of any of these attributes with a simple Boolean expression, for example:

```
if obj:attributes().folder then ... end
```

If the ShellObject represents a folder or a file, also returns the modification, creation and access dates. These are Time objects (described above). For files, also returns the file size. This is in kilobytes (floating point).

text = obj:displayname([type [, infolder]])

Returns the name of the ShellObject as a string in one of several formats. Type is a string, one of "normal", "forediting", "foraddressbar" or "forparsing" (default is "foraddressbar"). Infolder is a Boolean which, if true, limits formats which normally produce a complete path to the relative path within the containing folder.

res = dest:fileoperation(op [, spec] [, source] [, conf])

Performs bulk file operations on the destination folder represented by dest, or deletes, erases or renames the file or folder represented by dest. Op must be a string, one of "copy", "save", "move", "delete", "erase" or "rename". Spec is optional and is either a string or a Table which specifies the relative paths to the files and folders to be operated on. Source is required for "copy", "save" and "move" and must be a ShellObject. If spec is also supplied, source must be a folder. Conf is optional and specifies the level of user interaction as "none" (no interaction), "newfolder" (confirm folder creation), "delete" (confirm file or folder deletion), "progress" (give progress feedback only). The default is "delete" and each level includes the later ones.

NOTE: Folders are treated as files in all operations, so if a specification resolves to a folder, that folder and all contained files and sub-folders are operated on as a unit. Wildcards select folders as well as files.

The method returns Boolean true if the operation was completed successfully, false if the user cancelled the operation from any displayed UI element or an OS determined error number if the operation could not be completed for some other reason.

Spec is either a string or a Table of strings representing relative paths to files or folders (all strings must start with a backslash and include at least one further character). The string form is simply a shortcut equivalent to a Table with one entry. For "rename" this entry has the path of dest relative to its parent as the key and the string as the value. For all other operations, this entry has the string as the key and Boolean true as the value. The key strings specify the files to be operated on and the corresponding values (if they are strings) specify new names or relative locations for each key string. If the value is not a string, the files and folders retain the same name and relative location after the operation. An entry in the table can be inhibited by setting its value to Boolean false. Provided NONE of the keys have string values, wildcards may be used in the file or folder name part of the key strings.

Copy: If spec is absent, copies the source file or folder into the dest folder. If spec is provided, copies the specified files from the source folder into the dest folder. They will have the same relative path and name unless new ones are given in the spec table.

Save: As 'copy' except that in the event of a name collision, the copy is automatically given a new name (versioned).

Move: If spec is absent, moves the source file or folder into the dest folder. If spec is provided, moves the specified files from the source folder to the dest folder. The relative path and name will be the same in the dest folder unless new ones are given in the spec table.

Rename: Spec is required and source must be absent. As 'move' except that the old and new names in the spec table are both relative to the dest folder. If spec is a string, this is taken as a new name for the file or folder represented by dest itself, relative to its parent (so the string must still start with a backslash).

Delete: Source must be absent and spec is optional. If spec is provided, the files specified are removed from the dest folder and put in the recycle bin if possible. If spec is missing, the file or folder represented by dest is itself removed.

Erase: As 'delete' except that the recycle bin is not used and the files and folders are deleted permanently.

res = obj.execute([verb [, ...]])

Executes a shell verb against the ShellObject. Shell verbs are generally the first entries in the context menu for a ShellObject in Windows Explorer, although shell extensions may also provide verbs and place them anywhere on the context menu, or even not surface them to Windows Explorer. Verb is a string and if absent or an empty string, the default verb (shown in bold in the Windows Explorer context menu and executed on double-click) is executed. Verb is followed by an optional list of string flags from the set documented below.

Common verbs include "edit" and "print" for document files, "open" for executables, documents or folders, "explore" or "find" for folders, "runas" for executables (the "Run as administrator" operation in Windows Vista) and "properties" which launches the properties dialog for most object types.

The flags can comprise any set of the following string keys, specified in any order:

noinvoke - This flag uses an alternative method of executing the verb. Usually it is not necessary, however if a verb that is expected to work does not, it is worth trying again with this flag. Technically, this causes the verb to be recovered directly from the registry rather than using the IContextMenu interface of the item's shortcut menu handler.

nounicode – Normally the application is assumed to be Unicode, but this flag removes that assumption and may be necessary particularly for older applications.

noerrorui – Specifies that no UI dialog should be displayed in the event of an error.

ddewait – Another flag to try if things do not work as expected. Technically this specifies that any DDE transaction should be completed before the method terminates which may slow things down but make them more robust.

logusage – Specifies that this operation should increment the usage count of the application.

nozonechecks – Bypasses XP SP1 and later zone checks.

newconsole – Creates a new console for the process.

Operations

for child in obj(...) do ... end

Iterates the children of the current ShellObject. Code placed between the do and the end will be executed once for each child with the variable (child in this case) being a ShellObject representing that child. The call allows a variable number of string flags from the set “nofolder”, “nofile” and “nohidden”. By default, all objects are returned, “nofolder” filters out folders, “nofile” filters out everything except folders (usually this means files, but also special objects), “nohidden” filters out objects with the hidden attribute.

Table Class

The Table class of objects form a light-weight wrapper round the generic Lua table. No assumptions are made about the layout of the table and so any Lua table can be safely converted to a Table object. Note that the Lua # operator will not give a reliable count for entries in a generic table; the count method should be used instead.

Creation Function

tab = new.Table([table])

A new empty table is created by using this function without a parameter. If a Lua table is supplied, this table is converted into a Table object retaining its current contents.

Methods

tab:merge(tab2)

Copies all the entries in tab2 into the table.

tab2 = tab:invert()

Returns a new Table which has the keys and values in the original Table reversed (values become keys and keys values). This is useful for producing a reverse lookup table or index.

n = tab:count([limit])

Returns the number of entries in the Table. One of the limitations of the Lua table model is that no running count of entries is kept, so the implementation of a count of entries in a large table requires a full traversal. The optional limit parameter stops the count when it reaches this number, improving efficiency if an exact count is not needed (for example test for empty with tab:count(1)).

Operations

for key, val in tab() do ... end

This function call syntax is provided as a shortcut to pairs(tab) in the generic for statement.

for str in tab("") do ... end

The function call iterator syntax is extended to produce a stringized version of the table entries by passing a string parameter (which may, and should, be an empty string). The string produced is a valid fragment of a table initializer, for example ['key']=value;.

List Class

The List object assumes keys are integers (Number type) contiguous from 1 up to the number of entries. Values may be any type, and may be mixed. The standard # operator in Lua returns a reliable and efficient count for Lists.

Creation Function

lst = new.List(...)

Returns a List object. Any parameters supplied are added to the list in order.

Methods

lst:add(...)

If exactly one List object is supplied, the items in this list are added at the end of lst. Otherwise one or more parameters are added as new items to the end of lst. This is a shortcut for insert with an index of 0.

lst:insert(index , ...)

Inserts one or more items into the list. If exactly one List object is supplied, the items in this list are inserted in list order. Otherwise one or more parameters are inserted in parameter order. The first item is inserted at the specified index. Negative indexes are also supported with -1 being the last item in the current list, and 0 meaning to add the new items at the end of the lList. The original item at the specified index, and all subsequent items, are shifted after the inserted items.

lst:remove([from [, to]])

Removes the items starting at the from index and ending with the to index and shifts subsequent items down to fill the gap. From defaults to the last item and to defaults to the from item. Negative indexes are supported with -1 meaning the last item and so forth.

lst:sort([compare])

Sorts the List in place, changing the index numbers to reflect the sort order of the values. By default the Lua < operator is used to compare values, but optionally a compare function may be provided to give a different sort order. This function takes two values and returns Boolean true if the first is less than the second.

str = lst:concat([, sep [, from [, to]]])

Returns a string formed by concatenating the items in list order separated by the string sep which defaults to empty string. From specifies the index to start from (defaults to the first item). To specifies the last index to use (defaults to the last item).

There is also a tostring metamethod which uses concat with a separator “, “ and encloses the resultant string in brackets.

Operations

for val, key in lst([start] [, end]) do ... end

Function call syntax replaces standard Lua ipairs(lst). However note that the entries are produced value first, key (index) second, the reverse of ipairs. This simplifies the commonest case were the key is not required. Like ipairs, the entries are produced in key order from 1 up. Two optional parameters, start and end allow a subsection of the List to be produced by specifying the first and last index.

Set Class

Objects of the Set class represent a number of values which are either in or not in the Set. The values are stored as the keys of a standard table and the table value is always Boolean true. Therefore testing for membership of a Set resolves to simply testing the Boolean value of a table entry.

Creation Function

set = new.Set(...)

Returns a new Set object which either has no members or contains the members provided as parameters.

Methods

set:add(...)

If exactly one Set object is supplied, the members of this set are added to the set. Otherwise, the parameters are added to the set.

set:remove(...)

If exactly one Set object is supplied, the members of that set are removed from the set if present. Otherwise the parameters themselves are taken as members to be removed (if necessary) from the set.

str = set:concat([sep])

Returns a string formed by stringizing the members of the Set and concatenating them separated by the string sep (which defaults to an empty string).

There is also a tostring metamethod which stringizes the Set by using concat with a separator ", " and enclosing the result in square brackets.

Operations

==, ~=, >, <, >=, <=

The relational operators are overloaded to express the subset relationship. For example, a <= b is true if b contains all the members of a and zero or more additional members.

Delegate_List Class

Delegate_List is a specialisation of List in which the list members must be functions. It inherits all the methods of List but does not support the generic for iteration operation. Instead the function call operation is overridden to call all the functions in the list in turn.

Creation Function

dls = new.Delegate_List(...)

Returns a new Delegate_List object containing any functions that are supplied as parameters. Further functions can be added to the list using the add method inherited from List.

Additional Methods

dls:reverse([Boolean])

By default, the functions are called in list order. This behaviour can be reversed by first calling this method with no, or a Boolean true parameter. Original behaviour can be restored by calling with a Boolean false parameter.

Operations

dls(...)

Calling a Delegate_List object as a function calls each function in the list sequentially, passing each any parameters supplied to the call. The functions in the list should return nothing, or a nil as the first return, in order for normal processing to continue through the list. If a function returns one or more values and the first return is other than nil, the calling terminates, no further functions are called and the call returns the values returned by the terminating function.

Generic Object and Class System ('new' Library)

The generic object and class system used to create the Grunt built-in objects can also be used to create object classes, and objects from those classes, in scripts.

Functions

new.class(name [,source1] [,source2])

Creates and registers a new class with the name supplied as a string parameter. If this name is already in use, returns nil, otherwise returns the class metatable (ordinarily this can be ignored but in some cases it is useful to be able to make some further modifications to it). Source1 can be a string in which case it is the name of the parent class and the metatable for that class is used as a source for copying metamethods (implementing a form of inheritance), or it can be a table containing metamethods for the new metatable. Source2 can be another table containing metamethods which will override any with the same name from source1. After merging metamethods from these two sources, __type and __index fields are added overwriting any from the sources. __type is set to the name parameter, or if a parent class name is supplied as source1, to the name parameter prefixed to the name of the parent class and separated from it with an underscore.

The two source tables can contain any of the standard metatable fields documented in Section 2.8 of the Lua Reference Manual 5.1 plus any class specific methods. In addition, the metamethod __init may be provided (the purpose of this is described below).

obj = new.{class}(...)

obj = new.object("class" [, ...])

There are two alternative ways of creating a new object from an existing class. The first form is a shortcut to the second and is the form normally used. For example a List object with two entries can be created either:

```
obj = new.List("ham", "eggs")
```

Or

```
obj = new.object("List", "ham", "eggs")
```

Both methods work the same. The metatable corresponding to the class is recovered from internal storage and checked for an __init method. If there is none, a new empty table is created for the object, the metatable is attached to it and the table is returned. If there is an __init metamethod it is called and passed any additional parameters supplied to the new.object function. The metamethod is responsible for creating a new table and initialising it with any fields required. If the metamethod returns a table, new.object attaches the metatable and returns the table as the new object. If the metamethod does not return a table, new.object aborts object creation and returns nil. (The __init metamethod may alternatively return a Userdata rather than a Table, but this is not normally feasible for classes defined in Lua.)

obj = new.object(object)

The new.object function can also be used to make a copy of an existing object.

new.classes()

Returns a List object containing the names of all the classes available.

new.classof(var)

Returns the class name of a variable as a string, or nil if the variable does not contain an object.

John Hind 16th February 2009.