tkCoLa

Tk-based Command Language

Jason McDermott
JEMWorks Software, 2000
jemworks@jasonya.com
http://www.jasonya.com/jemworks

Table Of Contents

I. tkCoLa Overview

The **Tk Co**mmand **La**nguage (tkCoLa) is a Python module which allows rapid creation of GUI's for functions written in Python. It is essentially a simplified interface to the Tkinter module (which is in turn, an interface to the Tk/Tcl 8.0 library), that allows entry of values by the user and passage of the values to the specified functions in Python. This is especially useful for creation of programs which require a great deal of flexibility and are likely to be changed or added-to frequently. It also works well for creation of basic user interfaces, as long as you don't want to get too fancy.

Implementation of tkCoLa is in two parts. An implementation of the Command_Structure class provides the initialization and interface to the commands. Commands are specified in an external command file which employs a simple, fairly rigid syntax to define the user interfaces for particular commands. Through the Command_Structure commands can be listed in a list box or in a command menu. Commands can be grouped into functional groupings and this may be reflected in the list or menu.

II. Command Language Syntax

A. Overview

The syntax of the tkCoLa language generally follows the form:

```
keyword |field 1 |field 2 |field 3 …
```

where the keyword may be a global-type declaration, a command definition, or options which are contained by a command. A command is a container for all user input necessary for a particular function in Python. Therefore, a command will generally have a function call associated with it. This means that the user can invoke the command window, enter all the values required for running a function, then execute the function, from within the command structure. Command definitions will generally have the following layout, noting that the indentation shown is simply for purposes of clarity, it is not required in tkCoLa:

```
command |title |function
     command_modifier |f1 |f2
     optionA |f1 |f2 |f3 |f4
```

```
optionB |f1 |f2 |f3 |f4 …
     option_modifier |f1 |f2 …
```

Since the function is called in Python it must be specified as a Python call from inside the command structure. The command structure has a parent variable which refers to the object which called for its initiation. Function and variable references from inside the command structure take the form of **`self.parent.function`** or **`self.parent.variable`**. The exception is when a Python module has been invoked from inside the command structure and the function is in the imported module.

Commands are referred to by their titles and the titles must therefore be unique.

Required fields are denoted with an asterisk. Leaving non-required fields blank will give the default behavior.

B. Special Characters

|	: this character serves to separate fields for commands.
#	: specifies a comment line.
$	: used to indicate the continuation of an input line past a return. It also separates keyword assignments from a procedure specification and in certain cases separates attribute names from the name of the list or dictionary which contains them (see the Generator specifications, below for clarification). Also indicates the presence of a special value, for defaults or other fields, for example a variable reference instead of a simple string for the default field. Serves several other separation roles (see `text` option below for an example).

C. Global Declarations

import_file

```
import_file    |command_file
import …
```

Imports an external command file and appends it to this one.

NOTE: a warning will be issued to STDOUT if the imported command file contains commands which having non-unique titles but the old commands will be replaced with the newer ones.

`*command_file`: Valid TkCoLa command file.

import_module

`import_module |module_name`

`module …`

Imports a Python module into the command structure. The module can then be referenced from within tkCoLa by referencing the tkCoLa global variable with the name module_name (i.e. $Module_Name…).

`*module_name`: Name of a Python module (minus the .py).

prefix

`prefix |function_prefix`

Designates a global 'scope' prefix such that functions and variables beginning with a `'.'` will be preceded by the `function_prefix`. If no prefix is specified in a command file, the prefix defaults to `self.parent`. In the default case a function specified as `.my_function` would be interpreted as `self.parent.my_function`.

`*function_prefix`: a python object designation that can be used from inside a command structure (i.e. beginning with self.parent).

group_range

`group_range |group_name`

Specifies that commands following will be included in the listed group. The group does not need to exist already, if it does, the following commands will be

added to commands already in the group. The group_range declaration is only terminated by another declaration of group_range.

`*group_name`: Command group name.

set_var

`set_var |variable_name |value`

`var …`

Sets a command structure global variable with variable_name to the specified value. The value specification is evaluated by the Python parser so can be a list, or dictionary definition but must include quotes for string definitions. Global variables names preceded by a '$' can be used in default fields, function and callback fields, argument declarations for callbacks, and for setting tk configurations.

`*variable_name`: Name for the variable.

`*value`: Value for the variable in proper Python syntax.

D. Command Initialization

command

`command |title |function |event`

Begins a command definition. This will generally be followed by a number of options which can be set by the user and completed with a button (or buttons) allowing execution of the associated function. A command with no associated GUI options will be executed immediately and will not prompt the user for input.

Variables which do not require user input (i.e. static variables) may be specified by the following syntax:

`function$keyword=value, keyword=value, …`

The function may also be specified in 'shorthand' to use the scope prefix (see prefix, above) by using the syntax:

`.function`

Of course, the two forms can be combined.

NOTE: it is important to remember that everything following the `$` separator is interpreted literally by the Python parser so must have proper object specifications (i.e. `self.parent.object`) and must conform to Python syntax.

command example:

```
command |my command
     $ |.my_function$print='y',object=self.parent.myobject
```

(Note the alternative use of the $ character to continue the line)

`*title`:	Unique command title.
`*function`:	The Python function to be called by this command definition.

command_set

`command_set |title`

Begins the definition for group of GUI options which does not have an associated function call. For example a menu bar definition. A command_set is specified in the same way as a command.

`*title`:	Unique command title.

component

`component |title`

Begins a component definition. A component is a group of options which can be reused in any number of command definitions such as a button bar or menu definition. To use a component in a command definition use the `insert_component` option (see below). A component is specified in the same way as a command.

`*title`: Unique command title.

E. Command Modifiers

Command modifiers are declarations which follow a command declaration and modify the behavior of the command in various ways.

bind_all

bind_top

bind_all |event |callback
bind_top |event |callback

Binds the event to the callback for **all** widgets in the current command definition or to the **top**level window associated with the command.
Note: **bind_top** should be used with some caution since it will override (or add to) all event bindings for all widgets enclosed in by the toplevel window.

`*event:` Event binding.
`*callback:` Callback to bind event to.

Event Bindings

Event bindings in tkCoLa are specified similar to those in plain Tk. Below are listed the event bindings and forms recognized by tkCoLa. These definitions should not be enclosed in brackets (e.g. "**<**`Return`**>"** should be "`Return`"). If an event is enclosed in brackets it is interpreted by tkCoLa as a literal Tk event binding and will be handled as such.

Keys: Used by themselves or with modifiers (below).

`[letter/number/character]`	key press
`space/sp`	space bar
`return/ret`	Return key
`tab/tb`	Tab key

`escape/esc`	Escape key
`up/uparrow`	Up arrow key
`down/downarrow`	Down arrow key
`left/leftarrow`	Left arrow key
`right/rightarrow`	Right arrow key

Buttons: Used as "**`[button]-[123]`**" where 1,2 or 3 is the number of the mouse button.

`button/but`	Single button click.
`double/dbl`	Double button click.

Modifiers: Used as "**`[mod]-[key/button]`**". These can be combined to indicate multiple modifiers.

`alt`	Alt or Command key.
`ctrl/cntrl/control`	Control key.
`shift/shft`	Shift key.
`option/opt`	Option key (Macintosh).

label

`label |explanatory_text`

Allows an arbitrary amount of explanatory text to be associated with the command. Under default settings this text will be displayed in an outlined frame at the bottom of the frame enclosing the command GUI. Multiple label instances will simply concatenate the explanatory text.

`*explanatory_text`: Text to assist the user using the command.

static

`static |variable=value, variable2=value2, …`

This option allows variables that will not be altered by the user to be set (i.e. variables that will remain static in terms of the command). This can also be accomplished in the function field of the command definition (see above).

`*variable`:	Name of variable to be set.
`*value:`	Value (in proper Python) for the variable.

default_callback

default_callback |set_default_callback |refresh_function

Defines a callback that will set default values for the variables when the command is invoked. The refresh_function field may be used to optionally define a function that, when invoked will first cause the set_default_callback to be called. This option has not been rigorously tested and should be used with caution. See also the Special Command Structure Methods.

`*set_default_callback`: A valid Python function which accepts a variable, value_dict which is a dictionary of variable names (as keys) and associated values from the command's options and returns the same dictionary with appropriate default values for each variable.

`refresh_function`: Defines an existing Python function which will now cause the set_default_callback to be invoked each time the refresh_function is called. This option can be somewhat dangerous since the command_structure modifies the definition of the refresh_function to first invoke the set_default_callback then invoke the original refresh_function. The original refresh_function will be returned to its original state upon normal exit of the command.

post_command

post_command |function |pass_args_flag

Adds a function to the post_commands list. This is a list of functions which will be executed following the function defined by the command definition. This might be useful for updates, messages, etc.

`*function`: A valid Python function to be called after the command is finished.

`pass_args_flag`: If true the variables from the command will be passed along to the function, if false no variables will be passed. Default is true.

command_flag

`command_flag |evaluation_flag`

Allows the user to make the display and operation of the command contingent upon the state of the evaluation_flag. The command will be inoperative if the evaluation flag returns zero.

`*evaluation_flag:` A valid python method, function, or variable. If this flag is non-zero at the time of command execution, the command will operate normally. If the flag is zero, the command will not be displayed.

F. Options

Options follow a command initialization and define the user interface for the command, both in terms of information required of the user and graphic layout.

For options with callbacks, the option itself (which is a class instance, see below), can be passed as the variable `option`. For an example, see the Color Picker Example section, below.

Miscellaneous Options

insert_component

`insert_component |component_title`

Inserts the contents of the specified component. Component insertions are parsed during command structure initialization. This means that a component must be defined prior to its insertion in a command file.

`*component_title`: Title of a valid component already defined.

Formatting Options

row/column

row |frame_label |in_frame |padx |pady |side

column |frame_label |in_frame |padx |pady |side

Defines the start of a new row or column 'frame' for layout of options. Options following this definition will be arranged according to the rules of the format definition. In the case of rows, options are displayed starting at the left and are added to the right in a row arrangement. In the case of columns, options are displayed starting at the top and are added below in a column formation. Rows and columns are terminated by another row or column definition.

Layout of more complex interfaces often takes a little trial and error with regards to placement and use of the rows and columns. Look at the demo command files provided for examples.

`frame_label`: Label for the 'frame' defined by the row or column. This label needs to be unique only in the context of the command being defined. Defaults to unspecified.

`in_frame`: Valid label of a frame in the command which has already been defined. This allows for layouts like a row of columns or a column of rows. If no enclosing frame is specified the new frame will be enclosed by the frame defined last (if no frames have been specified, the new frame will be enclosed by the frame enclosing the command itself).

The special label **`_top`** refers to the frame which encloses the command.

`padx`: Space to be added between options horizontally. Default is 5 pixels.

`pady`: Space to be added between options vertically. Default is 5 pixels.

`side`: To which side of the enclosing frame will the new frame stick. **`top/bottom/left/right`** are valid values. Default for rows is top, columns is left.

set_tk_config/add_tk_config

`set_tk_config |tk_var:value, tk_var2:value2, …`

`add_tk_config |tk_var:value, tk_var2:value2, …`

Allows control over the behavior of the Tk widgets using their configuration keywords.

These declarations are sensitive to location in the command file. If they precede all command definitions (including components) they set a global Tk configuration, defining the default for all options. If they are used directly following a command declaration (but before any options) they modify the configuration for all of the options in that command. Finally, if they are used following an option they modify the configuration for that option only. The declaration `set_tk_config` will override all previous configurations and the declaration `add_tk_config` will add to any existing configuration with duplicate keywords overriding existing ones.

The command structure checks to see if the keywords are appropriate for each widget so no Tk errors should be encountered. This means that configurations for multiple widget types can be defined in a single global declaration. Some useful configuration keywords are listed below.

`tk_var`: Keyword of a valid Tk configuration option. Leaving this field blank will reset the tk_configuration to default parameters.

`value`: Value for the option, in proper Python (e.g. quotes surrounding strings, etc.).

Tk Configuration Options

`background` background color of widget.

`foreground` color of text, controls, etc. of widgets.

`font` font to be used for text.

`width`	width of widget (can be used to standardize a column of buttons, for example).
`height`	height of widget.
`callback`	Python function to be called when the widget is used. This should probably be used only for individual options.

Interface Options

In general interface options create a widget which allow the user to interact with the command. Many interface options have an associated variable which will have a value that the user determines. When a command is terminated by the user, the variables from all of the options will be passed as keywords to the function(s) associated with the command. Notes applicable to all interface options follow.

var_name field

The var_name is the name that will be passed as a keyword for the option. This means that it must fit the requirements for a valid Python keyword, i.e. not have any spaces or other special characters. In addition since Python prohibits passing redundant keywords, the var_name must be unique for the command. This is especially important to keep in mind when dealing with inserted components which have variables.

label field

The label field is a text field which serves to identify the function of the option to the user. Labels for buttons specify the text that will be placed on the button, etc.

var_type field

Specifies the type of variable for this option. Valid variable types are as follows:

Int/int/INT/I/i	integer type variable.
Float/float/F/f	float type variable.
String/string/S/s	string type variable.
list_int/L_I/l_i	*list of integers.
list_float/L_F/l_f	*list of floats.
list_string/L_S/l_s	*list of strings.

default field

Specifies a default value that will be assigned to the option when the command is invoked. Depending on the type of option the default value may specify the state of the widget or a default value of a type appropriate for the variable type specified. In the case of strings, no enclosing quotes are necessary for default values.

Most default values can be specified using either Python declarations or by specifying a previously defined tkCoLa global variable. In both cases these fields must be preceded by a '$'.

Option Examples

```
set_var |FILENAME |"default_file_name.txt"
command |Enter File Name |.open_file
     entry |file |File: |str |$FILENAME
     insert_component |command_bar
```

This example creates a command which prompts the user for a file name, where the default value is specified as a tkCoLa global variable. The set_var declaration establishes the tkCoLa global variable `FILENAME`. The command declaration begins the command definition. The `entry` declaration then uses the global variable `FILENAME` as the default value. Finally, a `command_bar` component is inserted to allow the user to close the window (see the section on Closing a Command, below).

```
command |Enter File Name |.open_file
     entry |file |File: |str
|$self.parent.file_str
     insert_component |command_bar
```

This example serves the same purpose as the previous example, however, the default file name is obtained by looking at the variable `self.parent.file_str`, which could be different every time the command is invoked.

entry

`entry |var_name |label |var_type |default`

Creates a text entry widget with a text label to the left of the entry field. The variable is set to the input value.

`*var_name`:	Variable name for the option.
`*label`:	Label to be placed on to the left of the text entry field.
`*var_type`:	Variable type of the option's variable. Supports list-type variables (see Option Generators, below).
`*default`:	Default value for the variable.

check

`check |var_name |label |var_type |on_val |off_val |default`

Creates a checkbox for entry of binary values. The label is placed to the left of the checkbox, though in future releases this should be mutable.

`*var_name`:	Variable name for the option.
`*label`:	Label to be placed on to the left of the text entry field.
`var_type`:	Variable type of the option's variable. Defaults to integer.
`on_val`:	Value given to the variable when the checkbox is checked. Defaults to 1.

`off_val`: Value given to the variable when the checkbox is not checked. Defaults to 0.

`default`: If true, the checkbox starts out checked, if false the box starts unchecked. Defaults to false.

link

link |on_list |off_list

Adds functionality to the checkbox definition which it must follow. Widgets in the option indicated by the variable names in the `on_list` are active when the checkbox is checked, while those indicated in the `off_list` are disabled (grey and unresponsive to the user).

`on_list:` List of variable names (separated by commas) indicating the options that will be enabled when the box is checked (and disabled when it is not) in the containing command definition.

`off_list:` List of variable names indicating options that will be enabled when the box is not checked (and disabled when it is).

scale

scale |var_name |label |type |from |to |inc |def |orient

Creates a scale widget.

`*var_name`: Variable name for the option.

`*label`: Label to be placed on to the left of the text entry field.

`*type`: Variable type of the option's variable (int or float).

`*from`: Starting value (lower bound) of the scale.

`*to`: Ending value (upper bound) of the scale.

`*inc`: Increment for the scale slider.

`*def`: Starting value for the slider.

`orient:` Orientation of the slider, vertical or horizontal. Default is horizontal.

rebutton

rebutton |var_name |label |callback |var_type |default
$ |pass_args_flag

Creates a button widget which invokes a Python function. The Python function callback should return a value which will be associated with the variable for the option. Note: rebutton is short for 'return-button'.

`*var_name`:	Variable name for the option.
`*label`:	Label to be placed on the button.
`*callback`:	Valid Python function to be called when the user presses the button.
`*var_type`:	Variable type of the option's variable.
`*default`:	Default value for the variable.
`pass_args_flag`:	If true, the variables associated with the entire command will be passed to the callback, if false, no variables will be passed. Defaults to true.

frame

frame |label |fill_callback |var_name |var_type
$ |get_value_callback

Creates an empty frame and calls the specified callback to fill it. Note that this option does not necessarily have a variable associated with it.

`*fill_callback`:	Valid Python function to be called to fill the frame. The function should take a keyword, 'frame' that gives the Tk frame to be filled. If the function returns any object or value it can be referenced by the variable `option.object`.
`*label`:	Label to be placed on the button.
`var_name`:	Variable name for the option.
`var_type`:	Variable type of the option's variable.
`get_value_callback`:	A valid Python function which will return the value for the variable.

radio

radio |var_name |label |var_type |default |orientation

Begins a radio-button definition. The radio declaration must be directly followed by at least two `radopt` declarations (see below).

`*var_name`:	Variable name for the option.
`*label`:	Label to be placed at the top of the radio buttons.
`*var_type`:	Variable type of the option's variable.
`*default`:	The default must be a valid label of one of the radio's `radopts`.
`orientation:`	Horizontal or vertical. Default is vertical.

radopt

radopt |label |value

Adds a radio button item to a radio definition.

`*label`:	Label for the radio button.
`*value`:	Associated value for this choice.

pop/pop_menu/popup_menu

pop |var_name |label |var_type |callback |default
$ |fill_callback |display_keys

pop_menu …

popup_menu …

Begins a menu definition for a pop-up menu which returns a variable. This definition requires either a fill_callback function to fill the menu, or that the declaration be followed by a list of menu_item declarations (see below).

`*var_name`:	Variable name for the option.
`*label`:	Label to be used as the menu title.
`*var_type`:	Variable type of the option's variable.
`callback`:	Python function to be called when a selection is made from the menu.

`default`: Label of the default menu_item.

`fill_callback`: Valid Python function to be called to fill the menu. The function will be passed the keyword 'menu' which refers to the Tk menu to be filled.

`display_keys:` If true, accelerator keys will be displayed in the menu for those items that have them specified.

menu_item

`menu_item |label |value |callback |key`

Adds a selectable item to a menu definition.

`*label`: Label for the menu item.

`*value`: Associated value for this choice.

`callback`: Python function to be called when this menu item is selected.

key: Defines a key-type event binding (see Option Modifiers, below for a list) which will invoke the menu command.

menu_sep/menu_separator

`menu_sep`

`menu_separator`

Adds a menu separator item to a menu definition.

select/multi_select

`select |label |list_var |select_action`

`multi_select …`

Creates a selectable list box filled with the contents of the list_var. The keyword select creates a single item selection box, multi_select creates a list box from which the user can make multiple selections.

`*label`: Label to be placed at the top of the list box.

`*list_var`: Variable name for the option.

`*select_action`: Python function that is called when a selection is made.

open_file

open_file |var_name |key |pattern |default |multi_file $ |path_file

Creates a box with selectable directories and files. The var_name keyword is returned as a string unless multi_file is true. If multi_file is true the var_name is returned as a tuple of the selected file names. This supports keys for saving default parameters and behaves in a similar manner to the FileDialog class from tkinter.

Note: in order for verification, filtering, and key usage to work use the **_fopen_bar** default definition for the command bar with the associated command.

*var_name:	The variable name for the option.
key:	A permanent key for saving file selection parameters input by the user. This may include the $ Python variable designations. Defaults to none.
pattern:	Pattern for the file filter. This may include the $ Python variable designations. Defaults to '*'.
default:	File name to initially fill the file selection box. This may include the $ Python variable designations. Defaults to none.
multi_file:	If true, users may select multiple files from the file selection list by clicking and dragging with the mouse, selecting a range with the shift key depressed, or selecting/deselecting individual files with the control key depressed. Defaults to false, single file selection.
path_file:	If this option is true then the entire filename, with path is returned for the variable. If it is false only the file portion is returned. Defaults to true.

save_file

save_file |var_name |key |pattern |default

Creates a box with selectable directories and files. The var_name keyword is returned as a string. This option supports keys for saving default parameters and behaves in a similar manner to the FileDialog class from tkinter.

Note: in order for verification, filtering, and key usage to work use the **`_fsave_bar`** default definition for the command bar with the associated command.

`*var_name:`	The variable name for the option.
`key:`	A permanent key for saving file selection parameters input by the user. This may include the $ Python variable designations. Defaults to none.
`pattern:`	Pattern for the file filter. This may include the $ Python variable designations. Defaults to '*'.
`default:`	File name to initially fill the file selection box. This may include the $ Python variable designations. Defaults to none.

Inert Options

Inert options have no variable associated with them.

text

`text |label |default`

Creates a line of text which can not be altered by the user.

*label:	Text to be displayed in the command layout. Inclusion of a variable name between '$'s allows for modification of the string by the default callback.
default:	Starting default value for the var_name (if any).

Text Example

```
text |The name of the file is: $file_name$.
|untitled.txt
```

This example will produce a text string with the default value replacing the `$file_name$` string in the label. This option is useful if a default callback is set. In this case the default callback can set the `file_name` keyword to display static text.

button

`button |label |callback |pass_args_flag`

Creates a button widget which invokes a Python function. This option returns no variable.

`*label`:	Label to be placed on the button.
`*callback`:	Valid Python function to be called when the user presses the button.
`pass_args_flag`:	If true, the variables associated with the entire command will be passed to the callback, if false, no variables will be passed.

menu

`menu |label |fill_callback |tearoff_flag |display_keys`

Begins a menu definition which does not return a variable. This is a simple menu which allows selection of items, each item having an associated callback. This definition must either have a fill_callback function or be immediately followed by a list of menu_item declarations (see pop_menu above).

`*label`:	Label to be placed on the button.
`fill_callback`:	Valid Python function to be called when the user presses the button. If no fill_callback is specified this declaration must be followed by a list of menu_items.
`tearoff_flag`:	If true, the menu will be a tearoff menu. Defaults to false.

`display_keys:` If true, the menu will display accelerator keys for the menu items that have them.

G. Option Generators

Option generators are special option definitions which allow creation of multiple individual options at the time of command execution based on a list- or dict-type variable. The existing Python list- or dictionary- type variable should be placed in the default field. The return variable will be in the form of a list or dictionary. Returned lists are identical order as the input list. Returned dictionaries have keys corresponding to the input dictionary. Each time the option is called it generates a list of **n** options of the specified type, each with a default value provided by the value of the list or dictionary member, where **n** is the number of members in the list or dictionary.

Lists (or dictionaries) of objects with attributes may also be used as long as the attribute to be used for the default value is specified. The attribute is specified using the following notation:

```
list_var$attribute
```

This requires, of course, that the list be composed of objects which have the specified attribute. See the example listed below for details.

button_gen

`button_gen |var |label |callback |type |list_var |pass_flag`

When the command is displayed, generates a series of rebutton options based on the values contained in the list_var.

`*var:`	Variable name for the returned list or dictionary.
`*label:`	Label for the buttons.
`*callback:`	Callback for the buttons.
`*type:`	Variable type for the individual returned values.
`*list_var:`	Python variable used to generate the buttons, either a list, tuple, or dictionary. If the variable consists of a list (or dictionary) of class instances, a specific class

	attribute must be specified using the **`list_var$attribute`** syntax.
`pass_flag:`	If true, the variable/value keyword pairs will be passed to the callback, if false they won't.

check_gen

`check_gen |var |label |type |on_val |off_val |list_var`

When the command is displayed, generates a series of check options based on the values contained in the list_var.

`*var:`	Variable name for the returned list or dictionary.
`*label:`	Label for the buttons.
`*type:`	Variable type for the individual returned values.
`*on_val:`	Value for the variable when the checkbutton is checked.
`*off_val:`	Value for the variable when the checkbutton is unchecked.
`*list_var:`	Python variable used to generate the buttons, either a list, tuple, or dictionary. If the variable consists of a list (or dictionary) of class instances, a specific class attribute must be specified using the **`list_var$attribute`** syntax.

entry_gen

`entry_gen |var |label |type |list_var`

When the command is displayed, generates a series of entry options based on the values contained in the list_var.

`*var:`	Variable name for the returned list or dictionary.
`*label:`	Label for the entry options.
`*type:`	Variable type for the individual returned values.

`*list_var:` Python variable used to generate the entry options, either a list, tuple, or dictionary. If the variable consists of a list (or dictionary) of class instances, a specific class attribute must be specified using the **`list_var$attribute`** syntax.

Generator Examples

```
entry_gen |list ||string |self.parent.my_list
```

This simple example will generate **n** entry options, corresponding to the members of `self.parent.my_list`, with blank labels. Each entry will start out with the value of the corresponding list member. The variable returned by this option (and passed as a keyword) will look like this;

`list=(input_string[0], input_string[1], … input_string[`**`n`**`])`.

```
check_gen |list |Flag |i |1 |0 |self.parent.ob_dict$use_flag
```

This example will generate **n** checkbox options each labeled "Flag **n**". The checkboxes will be on or off according to the value of the variable `self.parent.object_dict[`**`n`**`].use_flag`.
The variable returned will look like this;

`list={'key0':input_value[0], 'key1':input_value[1], … 'key`**`n`**`':input_value[`**`n`**`]}`

H. Option Modifiers

bind

`bind |event |callback`

Adds an event binding (see **`bind_top`** under ***Command Modifiers*** above) to the option which it follows.

`*event:`	tkCoLa version of an event to be bound.
`*callback:`	Callback to bind the event to.

callback

`callback |callback |pass_flag`

Adds a callback to an option. Every time the option is accessed by the user the callback will be called.

`*callback:`	Python method or function to call each time the option is changed by the user.
`pass_flag:`	If true all the keyword/value pairs from the command will be included in the callback. Default is true.

set_state

`set_state |option_state`

`disable(d)`

Sets the initial state of the option, normal or disabled. If the option is set as disabled the user will not be able to interact with it and the text in the option will be gray. This is especially useful when coupled with a check option having a callback which toggles the state of this option. See section IV, below for an example. The keyword, `disable` (or `disabled`), is an alias to set_state and disables the option.

`option_state:`	Acceptable values are true, false, normal, or disabled.

default_color

`default_color |where`

A modifier specifically for setting the color of an option's widgets based on the options value or default. This means that the option must be returning a valid Tk color name in the form of a string. See section IV, for an example of usage.

`where:`	Set the color to foreground or background of widgets. Default is background.

III. Command Class Descriptions

A. class ***Command_Structure***

B. class ***Command_Record***

Base class for command definitions. Command options are stored as a list (`opt_list`).

`title`	Unique title for the command, used as a reference as well as an explanatory title for users.
`command`	String containing the function field from the command definition.
`parent`	Python reference to the parental structure (i.e. the parent of the command manager).
`manager`	Python reference to the command's command manager.
`mem`	If true, default values will be remembered from one invocation of this command to the next (in the case of simple default declarations).
`echo`	If true, commands will be printed to the screen when they are executed. Good for debugging.
`tk_config`	tk_config dictionary to be applied to the options in this command.

C. class ***Option***

The base class for all interface options, aside from the special cases of Generator Options which create a list of options managed by a Generator_Option class.

Attributes

`type`	The type of option, this will reflect the option's tkCoLa keyword (e.g. entry, button, frame, etc.).
`var_name`	The name of the option's variable, if any.
`label`	The option's label.
`var_type`	The option's variable type (a Python type, not a string).
`default`	The default value for the option.
`variable`	The user input value to be returned with the variable.
`command`	The option's parent command instance.
`widgets`	A dictionary of the tk widgets associated with the option.
`tk_config`	The option's tk_config options (in the form of a dictionary).

Methods

`toggle`	Toggles the activation state of the widgets in the option.
`get`	Returns a Tuple containing the option's variable name followed by the value `option.variable`.
`update_default`	Updates the value for the option from the default value.
`draw_option`	Given a frame and configuration options (tk style), draws the option in the frame.

IV. Use of the Command Structure

A. Initializing a Command Structure

First make sure that the Commands module has been imported in the program. Then, to initialize the command structure, create an instance of the Command_Structure class passing the name of the command file to be used as the first argument and being sure to pass the controlling instance (self) as the parent. Be sure to make a permanent reference to the command_structure if your program will need to reference it directly.

Command Structure Example

```
# make the command struct
command_file = COMMAND_PATH + 'my_commands.com'
self.command_struct = Commands.Command_Struct(command_file,
                                                parent=self)
```

B. Adding a Command Menu

The command structure method `menu_init` can be used to initialize a command menu.

method Command_Structure.menu_init

Given a frame initialize a command menu which allows the user to choose from commands contained in the menu_groups.

Command menus can be initiated from a command file (say a menu bar command_set) by setting the fill_callback of a frame option to the menu_init method. From inside a command file the CoLa shortcut listed below can be used instead of the full definition, `self.parent.command_structure.menu_init`.

CoLa shortcut: `menu_init`

Options

`*frame/menu_frame`	The Tk frame that will hold the command menu.
`menu_groups`	A list of groups which will initially be included in the menu.
`menu_title`	The title to give the menu, defaults to 'Commands'.
`menu_type`	'normal' or 'cascade'. If normal, all command groups will be listed linearly in the menu. If cascade, each group will have its own cascading menu with associated commands. Defaults to normal.
`menu_select_color`	Color for the currently active command to be listed in the menu, defaults to blue.

`command_frame`	Frame to display requested commands. Default is to open a new Tk window for the commands.

Menu Example

```
command_set |menu_bar
        row |||0|0
        menu |File
                menu_item |Load File ||.open_file
                menu_item |Save File ||.save_file
                menu_item |Save File As... ||.saveas_file
                menu_item |Close ||.close
        frame |command_menu
                $ |.command_struct.menu_init$menu_groups=('gr1, 'gr2'),
                $ menu_type='cascade'
```

This example illustrates how to make a command menu from within a command file. The example above first defines a row with no padding (so that the menu buttons line up properly) then defines a file menu and finally defines a frame which has as its fill_callback a call to the menu_init method.

method Command_Structure.menu_set_groups

To change the groups associated with the particular command menu use the command structure method `menu_set_groups`. Calling this method forces an update of the command menu specified.

CoLa Shortcut: `menu_groups`

Options

`*menu_title`	The menu title of the command menu to be modified.
`*menu_groups`	A list of the groups to be included in the command menu.

C. Opening a Command

method Command_Structure.manager_window

Opens a command in a new window or the frame provided.

CoLa Shortcut: `manager_window`

Options

*command_title	Valid title of a command present in the command_structure
	-or-
*command	Python reference to the actual command instance. This can be obtained using the find_by_title method explained below.
frame/command_frame	Frame for the command to be opened in. Default is to open a new window.
menu_title	Menu that this command is associated with. This is mainly for internal use but might come in handy.
window_title	Title for the command window.
window_config	A valid set of options (either keyword or dictionary style) passed to the tk Frame instance for the command window. This can be used to set background color, border width, etc.
data	The data keyword can be used to pass any Python reference to the command. The object or variable can then be referenced as `command.data` (from the opened command this is of course, `self.data`). This can be used for a variety of purposes involving one

	command window calling another. See the Color Picker description below for an example.
data_process	A Python method which either takes the data above, modifies it in some way and returns the modified data or if no data is provided, generates the data that will become `command.data`.

D. Closing a Command

Closing a command window is generally up to the user. It is convenient to assign each of the methods below to a button and make this into a component (see Example following).

method Command_Structure.manager_ok

Closes the current command and passes its associated variables to the command's function(s).

CoLa Shortcut: `ok`

Options

destroy	flag, true if the frame should be destroyed, false if the frame should be hidden, not destroyed.

method Command_Structure.manager_cancel

Closes the current command but does not invoke the command's function(s).

CoLa Shortcut: `cancel`

Options

destroy	flag, true if the frame should be destroyed, false if the frame should be hidden, not destroyed.

method Command_Structure.manager_apply

Invokes the command's function(s) but doesn't close the associated window.

CoLa Shortcut: `apply`

Buttons Example

```
component |_command_bar
     set_tk_config |font:bold
     column |_command_bar |_top | | |bottom
     row | |_command_bar |5
     button |OK |ok |no
     button |Apply |apply |no
     button |Cancel |cancel |no
     row |rest |_top |||top
     column ||rest
```

This example creates a reusable component which consists of three buttons. An OK button, an Apply button and a cancel button. The text for the buttons is set to bold using the `set_tk_config` declaration. The first column establishes that the bar will be located at the bottom of the top frame. The row declaration assures that the buttons will be arranged horizontally. Finally the row and column at the bottom re-establish the normal function for the remainder of the command so that the `_command_bar` can be included at the start of the command definition. The `_command_bar` component is included in the default definitions for tkCoLa (see Default Definitions, below).

E. Special Command Structure Methods

Several other useful methods are also available.

method Command.command_default_object

When used as the Default_Callback, this method provides a way of linking objects with attributes to the default values of corresponding options. Objects are passed as arguments and values attribute names which match variable names of the command options will be used as the default values for the option.

CoLa Shortcuts: `command_default_object`
`cdo`

Options

*[arguments] An arbitrary number of objects with attributes that will be searched in order for matches to option variable names.

method Command.command_return_object

Matches attribute names in the objects passed as arguments with the keywords passed from the command and sets the corresponding attribute values to the value passed in the keyword. Provides an easy way to implement an class editor by simply matching the attribute names with the option variable names.

CoLa Shortcuts: `command_return_object`
`cro`

Options

*[arguments] An arbitrary number of objects with attributes that will be searched in order for matches to option variable names.

V. tkCoLa Examples

This section gives a tutorial of some basic examples of how to use the tkCoLa language. The Color Picker Example, below, provides a more complicated example.

A. A Simple Command Example

Say you have a method in your application which prints out information (`print_info`) about the user, and you wish to prompt the user for this information (name, age, PIN) before calling this method. We'll assume that your application has a command structure and a way for the user to interact with commands (like a menu). First we'll create the `print_info` method, then the command to access it, Input Information.

```
class Application:
…
     def print_info(self, name=None, age=None, PIN=None, **kw):
          print "User Name: %20s" % name
          print "Age:       %5i" % age
          print "Now I have your PIN! %s" % PIN
```

This is a Python method which should be included under your main application. It takes three keywords, name, age and PIN, and it can also handle any number of other keywords. This is important since the command structure will always pass back more keywords than are included with your command. It simply prints out a silly message using the input information. Using only Tkinter, prompting the user for input would require the use of a dialog class and writing who-knows-how-many more lines of code for your application. Using tkCoLa it simply requires the following tkCoLa command definition.

```
# A tkCoLa Command to allow user input to the method print_info
command |Input Information |.print_info
     label |Trust Me!
     entry |name |Your Name: |S |Bob Smith
     entry |age |Your Age: |I |35
     entry |PIN |Your ATM PIN: |S |0000
     insert_component |_command_bar
```

Place this command definition in your tkCoLa command file for this application (the one used by the application's command structure). Make sure that the command is in a group which will be accessible to the user (through a Command menu, etc.). The user may now invoke the command, input the information, press the Ok button (included with the `_command_bar` component) and watch the fun.

This is a very simple command definition, the defaults for the entry options will always start out with the same values. Note that if the memory option is enabled for your command structure the command will remember the values last input into the command (during that session of the application) and will use those as defaults.

B. A More Complex Command Example

You have an application which keeps a database of user information as a list of individual records and it has some kind of selection method for the list. This example explains how to use tkCoLa to create a record editor for the database. The application variable `selected_record`, refers to the currently selected Record instance of the database list. To make things clear, we'll first look at the class Record which holds information then look at how to use this to create a command.

```
class Record:
     def __init__(self, name=None, age=None,
                  relation=None, terms=None):
          self.name = name          # name
          self.age = age            # age
          self.relation = relation # how do we know them?
          self.terms = terms        # are we on speaking terms?
```

So any instance of the class Record will have the attributes listed above and the application variable `selected_record` is such an instance. Next, we'll create a command to allow editing of a Record class instance.

```
# Command to allow editing of the selected_record,
#         a Record class instance.
```

```
command |Edit Selected Record
                $ |command_return_object$parent.selected_record
     label |Edit Member Record
     default_callback
                $ |command_default_object$parent.selected_record
     entry |name |Member Name: |S |Bob Johnson
     entry |age |Member Age: |I |25
     radio |relation |Relationship |S |No Relation
          radopt |No Relation |none
          radopt |Relative |rela
          radopt |Friend |frnd
          radopt |Business |busn
     check |terms |Speaking Terms? |S |yes |no |on
     insert_component |_command_bar
```

This example illustrates how tkCoLa can be used to obviate the need for an associated Python method in your application.

First notice that the command defines four options and each option has a variable which matches the attribute name of the four Record attributes. Next, the `default_callback` is set to `command_default_object` and the application's `selected_record` is passed as the argument. This means that when the command is invoked it will fill the defaults for each option by looking through the `selected_record` object and matching variable names with attribute names.

To finish, the function which is called when the user presses the Ok button is the `command_return_object` method which essentially does the reverse of the `command_default_object` method. It looks through the `selected_record` and for variable names and replaces values of the object with those input by the user.

C. The Application GUI Example

This example demonstrates how to use tkCoLa to do the GUI layout for an application, including menu's, a button bar, status line, and an application defined frame.

D. The Color Picker Example

This example illustrates how to set up a command that allows the user to choose colors for several different features of a document. It employs several features of the command structure to accomplish this in a user friendly way.

The Color_Picker class is included in the Commands module and is an example of how a user defined widget (-type thing ;) can be used in conjunction with the tkCoLa to enhance the GUI.

The command will display several buttons. When pressed, each of the buttons will open a color picker command, which will return the picked color to the original command window.

The _Color Picker command is included in the tkCoLa default definitions (see Default Definitions, below).

class Color_Picker

A class which creates a widget to allow the user to choose from a panel of color boxes or to enter a valid color name.

Options

`color_map`	A list of valid tk color names to be displayed as boxes. Defaults to a standard color map defined in the Commands module.
`x_box`	X-dimension (in pixels) of each color box. Defaults to 10.
`y_box`	Y-dimension (in pixels) of each color box. Defaults to 10.
`outline`	If 1, outline the each color box in black. Defaults to true.
`rows`	Number of boxes displayed on each row of the color picker. Defaults to 15.
`show_current`	If 1, display the color which the mouse is currently over in a separate color box. Defaults to 1.
`name_box`	If 1, allow the user to enter their own color names in a text entry box. Defaults to 1.

`show`	If 1, on class instantiation, make and show the color picker (requires that a frame be specified). Defaults to 1.
`frame`	Tk frame in which to create the color_picker. If not provided the `make` method must be called to show the widget.
`color`	Tk color name for initial default color selection. Defaults to black.

Methods

`make`		Draws the Color_Picker widget in the specified frame with the specified default color. It is not necessary to call this method if the widget was shown during class instantiation.
	`frame`	Tk frame to draw the widget in.
	`color`	Initial tk color.
`get`		Returns the currently picked (or input) color name. Warning: this method does not check to see if the name is a valid tk color.

Creating the _Color_Picker Command

First we'll make a command which will display the color_picker widget.

```
command |_Color_Picker |command_return_object$self.data
     default_callback |command_default_object$self.data
     frame |color_picker
                    $
                    |Color_Picker$x_box=15,y_box=15,rows=11
                    $ |variable |string
                    $ |option.object.get
     row
```

```
insert_component |_simple_bar
```

The command declaration begins a command definition for our window. Note that the function that is associated with the command is the `comand_return_object` with the object passed being the command's `self.data` variable. This suggests that we will pass some object with attributes to this command from the calling command. The `default_callback` declaration sets the command's default callback to `command_default_object`, with the `self.data` as the argument. Using these two methods (`cro` and `cdo`) together is an easy way to allow editing of an object (or objects). In this case the `self.data` object.

The frame declaration defines a frame called color_picker, which has a string-type variable with the name `variable`. This implies that the object which is passed to `self.data` has an attribute, `variable`, we'll see why this is the case in the next step. The `fill_callback` for the frame creates an instance of the Color_Picker class and sets the x_box and y_box options to 15 and the number of boxes in each row to 11 to make it look nice. It leaves the color_map as the default which means that the color_map defined in the Commands module is used. Finally, the `get_value_callback` is referenced by the method `option.object.get`, since the color_picker instance is stored in the `option.object` variable (see the frame option above).

Finally, Ok and Cancel buttons are provided to allow the user to close the window.

The Example Command

Creation of a command which uses the Color_Picker command described above is simple, but requires a little explanation. Here is an example of a command with two color_picker buttons. In this case the function self.parent.set_colors takes the returned keyword/value pairs (back_color=[some_color], text_color=[some_color], border_color=[some_color]) and sets the appropriate variables in the document and presumably forces an update of the document in some way to display the new colors.

```
command |Set Document Colors |.set_colors
     default_callback |cdo$parent
```

```
label |Press the buttons to set colors for the
label |indicated document features.
rebutton |back_color |Background
     $ |manager_window$command_title='_Color_Picker',
     $ window_config='def', data=option |string |black|no
     add_tk_config |width:14
rebutton |text_color |Text Color
     $ |manager_window$command_title='Color_Picker',
     $ window_config='def', data=option |string |white|no
     add_tk_config |width:14
rebutton |border_color |Border Color
     $ |manager_window$command_title='Color_Picker',
     $ window_config='def', data=option |string |red|no
     add_tk_config |width:14
insert_component |_command_bar
```

The default callback is set to the command_default_object method, with the argument passed being `parent` indicating that the parent has the attributes corresponding to the three variable names of the command. The label explains to the user what to do with the buttons.

The `rebuttons` have different variable names and labels but the remainder of the declarations are all identical (with the exception of the default values which are overridden by the default callback anyway). The `rebutton's callback` is set to open the command we defined above, _Color_Picker, and it passes the option itself as the data keyword. Since the `option` has the attribute `variable` (the value which will be returned, not the variable name) the _Color_Picker window can access this to set it's default value and will modify the `option.variable` when Ok is pressed.

The standard _command_bar component is inserted at the end of the command to allow the user to terminate the command. The add_tk_config declarations following the rebutton declarations ensure that the buttons will all have the same width and so will line up nicely.

VI. Default Definitions