ConfigTreeView Documentation

Release 0.1.3

Wesley Hansen

Jul 19, 2017

Contents

CHAPTER 1

Contents:

About ConfigTreeView

This is an implementation of a GtkTreeView in python(using pygtk) that allows for easy, fast, and dynamic setting up of a TreeView, its TreeViewColumns and CellRenderers. This ConfigTreeView can create a TreeView with all its properties initialized through the use of a simple config-type file. This config file can be in a python dictionary format, or even read in from a JSON object.

The Why (Why use a ConfigTreeView?)

- The ConfigTreeView was designed in such a way to abstract the developer from having to set up indices for how a ListStore row of data should look. The config file creates an easy way to do it and allows you to supply a row of data in python dict form(or a JSON) while initializing all the properties, columns, cell renderers that could possbily be used in creating a TreeView.
- Eliminates the several lines of code it takes to initialize a TreeView. A TreeView is a very useful but also very complicated widget in the gtk arsenal and this implementation takes away that complication.
- Useful for data sets that could change frequently without having to go in and change the code.–This is actually the use case that I ran into at my place of work that inspired me to create the ConfigTreeView. We have an application that many people use in the office that is connected to a server. The application gets all of the data from the server and displays it in a GtkTreeView but the data could change in the near future as we may need to display new columns or different formats of data in the same TreeView so we wanted a system set in place that could allow for us to change the data the server was sending without having to go in and change the code in the clients(i.e. the TreeView) in order to properly display the newly changed data. With a ConfigTreeView you can do just this: the server can supply a config structure to initialize the clients, eliminating the need for changing the client code.

How to use it

- It's easy! All you need to do is create a config file(either as a python dict in a .py file or as a JSON file).

- Then with a config file, you're ready to create a ConfigTreeView:

```
from config_treeview import ConfigTreeView
#Import the config structure(it's a python dict named config)
from myconfigfile import config
#Create a ConfigTreeView using config as the configuration structure
treeview = ConfigTreeView(config)
#Apply the config structure to finish initalizing the TreeView
treeview._apply_config()
```

More About ConfigTreeView

The `ConfigTreeView` is an implementation of `GtkTreeView` that has its widgets, their properties, and attributes set via a configuration structure. Because of the nature and design of this config structure, a very simple, intuitive, and flexible data model has been created that both define the look and feel of the treeview, but also explicitly detail the shape and look of the data that the `ConfigTreeView` is expecting.

The `ConfigTreeView`'s data model is indirectly defined in the `index-names` key in the config structure. This key has a dictionary as its value that looks something like this:

```
{
  'cell-bg-index': bool,
  'market':{'pixbuf': "gtk.gdk.Pixbuf"},
  'status':[{'markup':str},{'markup':str}],
  'name':{'markup':str},
  'address':{'markup':str},
  'contact':{'markup':str},
  'payment':[{'markup':str}, {'markup':str}],
  'shipping':{'markup':str},
  'comments':{'markup':str},
  'placed':{'markup':str},
  'id': str,
}
```

Each inner-level key represents a new index of the defined type in a data row that will eventually be inserted into a GtkTreeModel.

The Data Model

The ConfigTreeView data model refers to what structure the incoming data must look like in order for the data to be properly formatted and displayed. The data model is indirectly defined by the `index-names` dictionary defined in the configuration structure. What this means is that the `index-names` dictionary is also the same structure that the data model should follow, replacing the types that were defined for the actual values at that particular index. An example data structure for a single row of data following the data model defined above:

```
{
  'cell-bg-index': True,
  'market':{'pixbuf': "/images/market1.png"},
  'status':[{'markup': "Current Status"},{'markup': "Shipped"}],
  'name':{'markup': "Darth Vader"},
  'address':{'markup': "The Death Star"},
  'contact':{'markup': "867-5309"},
  'payment':[{'markup': "Visa"}, {'markup': "$100.00"}],
```

```
  'shipping':{'markup': "Shipped"},
  'comments':{'markup': "No comments"},
  'placed':{'markup': "2012-06-19 12:08:33PM"},
  'id': "123435252335",
}
```

But the data model also is a little more flexible than this. There may come a time when you don't want to display all information for a row. With this data model you only need to define the indices that you plan on using. So a more minimal row of data could look like this:

```
{
  'name': {'markup': "Darth Vader"},
  'address': {'markup': "The Death Star"},
  'status': [{'markup': "Current Status"}, {'markup': "Shipped"}]
}
```

The Config Structure

We needed something that could initialize the treeview's gui components and their properties(TreeViewColumn, CellRenderer, and the TreeView itself), but also have an insight into what each renderer is expecting of the data(for setting up cell-specific renderer properties, or which data it's going to display) without having to specifically define the actual indices into the data that these components are searching for(the treeview will be able to handle that internally upon initialization using the config file)

Complete Structure, with every possible key:

Note: This structure details every possible key or key arrangement for the sake of it. It can appear to be complex and overwhelming at first glance, but don't fret. If a key that isn't required isn't used at all for the ConfigTreeView you want to build, then don't have to include that key. This applies to top-level keys as well. This functionality allows the user to rapidly and easily build TreeViews.

```
{
        "treeview":{
                "properties":{ #This struct will contain all treeview properties---
↪meaning things that can be set using `treeview.set_property( property, value)`
                        "rules-hint": True,#<---A property of GtkTreeView
                        "headers-clickable": True,
                },
                "args": [ #Positional arguments to send to a custom ConfigTreeView--
↪can be indices, or custom values
                        "$index.record_open_bool", #This is how you assign an index␣
↪as an argument to the ConfigTreeView
                        "$index.column_one.1.markup", #An example assigning an index␣
↪from a column that has multiple renderers--a number in a 'dotted' string represents␣
↪the index in the list of renderers
                ],
                "kwargs": {}, #Keyword arguments to send to a custom ConfigTreeView--
↪same rules as "args"
                'bg': None, #Set background color, bg-even and bg-odd override this␣
↪option
                'bg-even': '#A2C879', #Set even-row background color...must be␣
↪accompanied by bg-odd
                'bg-odd': '#6794AB', #Set odd-row background color...must be␣
↪accompanied by bg-even
```

```
                'selection-mode': 'SELECTION_SINGLE', #The selection mode of the␣
↪TreeView...must be one of SELECTION_NONE, SELECTION_SINGLE, SELECTION_BROWSE,␣
↪SELECTION_MULTIPLE
                'selection-color': '#bfd3e7', #Change the color of the selection bar--
↪value can be a valid color name or hex string
        },
        "treemodel": { #You can tell the TreeView to build you a custom TreeModel(via␣
↪the get_treemodel() function)
                "module": "package.subpackage.module", #The package/module location␣
↪of the class to import
                "class": "CustomTreeModelClass", #Your custom GtkTreeModel␣
↪implementation
                "args": [], #The positional arguments to send to the TreeModel's _
↪handle_args method
                "kwargs": {}, #The keyword arguments to send to the TreeModel's _
↪handle_args method
        },
        "index_names":{ #Maps the given index into the type of data it will store
                "record_open_bool": bool, #This is how you define a type of 'bool'␣
↪for an index that will be used by more than one column's renderer
                "order_status":{ #This is how you specifically define types for a␣
↪certain column(that contains only a single CellRenderer)
                        "markup": str, #The `markup` property of the cell renderer
                },
                "column_one":[ #This is how you define types for columns with␣
↪multiple renderers
                        { "markup": str, "cell-background": str},
                        { "pixbuf": "gtk.gdk.Pixbuf"},
                ],
        },
        "column_order": ['column_one', 'order_status'], #Define the order the columns␣
↪are appended in the treeview. List the column names in the order you want them␣
↪appended to the TreeView
        "macros": { #Define macros--a convenience for assiging the same properties to␣
↪multiple columns or renderers
                "col-default": {
                        "expand": True,
                        "resizable": True,
                        "clickable": True,
                        "reorderable": True,
                }
        },
        "columns": {
                "column_one": { #A column named "column_one"--the name is used when␣
↪determing: where columns go in the TreeView, which indices a column's renderers␣
↪will use, and their types
                        "macros": ["col-default"], #Assign a macro( by name )
                        'header': { #This struct contains information to create the␣
↪header for this column. A header can be a string or a custom widget. Defaults to do␣
↪nothing
                                'title': 'Column1', #A simple text label displaying␣
↪the title(This is a fallback to the custom widget)
                                'module': None, #The module that contains the header␣
↪widget, None defaults to the gtk module
                                'class': 'Button', #A string of the class( must␣
↪inherit gtk.Widget), to set the header widget to. None defaults to gtk.Label(If
↪'title' key isn't present)
                                'args': [], #Positional arguments to pass to 'class'
```

```
                                'kwargs': {}, #Keyword arguments to pass to 'class'
                        },
                        "properties": { #This struct will contain all TreeViewColumn␣
↪properties that can be set using treeviewcolumn.set_property(property, value)
                                "resizable": True,
                                "visible": True,
                                "max-width": 100,
                        },
                        "renderers":[ #Create the Renderers( use a list if multiple␣
↪renderers in one column)
                                {
                                        'pack': 'pack_start', #Either 'pack_start' or␣
↪'pack_end', if this is None, defaults to pack_start
                                        'expand': True, #Sets the packing method␣
↪'expand' property, if this is None, defaults to True
                                        'module': None, #The module that contains the␣
↪renderer, None if it's in the gtk module
                                        'class': 'CellRendererText', #A string name␣
↪of the  CellRenderer class to use( must inherit gtk.CellRenderer )...if this is␣
↪None, defaults to gtk.CellRendererText
                                        'args': [], #Positional arugments to pass to␣
↪'class', if None, don't pass args to 'class'
                                        'kwargs': {}, #Keyword arguments to pass to␣
↪'class', if None don't pass kwargs to 'class'
                                        'properties': { #This struct contains all␣
↪properties that each row will have by using cellrenderer.set_property( property,␣
↪value), if 'properties' doesn't exist(or None), then no props set.
                                                'height': 25,
                                                'xpad': 5,
                                                'font': 'Times New Roman 13',
                                        },
                                        'indices': { #This struct contains all␣
↪properties that are set from the treemodel data...using column.add(attribute,␣
↪property, index), None sets no attributes
                                                #These indices are determined at␣
↪runtime when the ConfigTreeView is initialized.
                                                #You can either set them by assigning␣
↪them a name( if one index will be shared by multiple renderers)
                                                #       NOTE: You must assign by a␣
↪name that exists in the 'index_names' key of this configuration file
                                                #Or you can set them by setting a␣
↪property value to True, telling the TreeView that this one is unique
                                                "markup": True, #This␣
↪CellRendererText needs its own 'markup' index in the data model
                                                "cell-background-set": "record_open_
↪bool"       #All columns will share this index

                                        }
                                },
                                { #The second renderer in the "column_one" column
                                        'class': 'CellRendererPixbuf',
                                        'indices': {
                                                'pixbuf': True, #This␣
↪CellRendererPixbuf needs its own 'pixbuf' index in the data model
                                                'cell-background-set': 'record_open_
↪bool', #All columns will share this index
                                        },
```

```
                                },
                        ],
                },
                "order_status": { #Another column, this one named "order_status"
                        'header': {
                                'title': 'Order Status',
                        },
                        'properties': {
                                'resizable': True,
                        },
                        'renderers': { #A single renderer can be defined as a single
↪dict instead of as a list
                                'markup': True, #This CellRendererText needs its own
↪'markup' index in the data model
                                'cell-background-set': 'record_open_bool'#All columns
↪will share this index
                        },

                }
        }

}
```

HOW-TO: Create a config file

This document details everything you need to know about creating a config file for use with a ConfigTreeView. First I'll say a few things about how a config file is made.

Making a config file

-The config file is represented as a dictionary in python. Because of this, you can either create a python dict in a separate .py file or create one on the fly within your code...OR...you can create the dict in a JSON document...and simply give the path to the JSON file in the constructor to the ConfigTreeView where you'd supply the python dict normally.

Basic Structure

-Here's a look at the top-level keys in the config dict:

```
{
        "treeview": {},
        "treemodel": {},
        "index_names":{},
        "column_order":[],
        "macros": {},
        "columns":{}
}
```

The *treeview* key(optional):

The *treeview* key(this key is not required, unless you're setting properties): This key sets up everything dealing with initializing the TreeView container. This includes setting gtk properties, some style options and also supplying custom positional and/or keyword arguments to your custom ConfigTreeView.

example:

```
{
        "treeview": {
                "properties": {
                        "rules-hint": True,
                        "headers-clickable": True
                },
                "args": [],
                "kwargs": {},
                "bg": "green",
                "bg-even": "red",
                "bg-odd": "yellow",
                "selection-mode": "SELECTION_SINGLE",
                "selection-color": "blue"
        }
}
```

The *properties* key defines gtk properties for the TreeView as found in the pygtk documentation: http://www.pygtk.org/docs/pygtk/class-gtktreeview.html Any of the properties listed here can be defined in the "properties" key. An entry in the *properties* dict takes the following form:

```
"gtk-property-name": "value"
```

Note: if the value is not a native python type(ex: a gtk object), this is not currently supported if you define the config file in a JSON as primitive types...but only a few properties are like that. If you need to use these types, consider it good practice to define the config structure in it's own python module. Then you can import all the modules you need.

The *args* and *kwargs* keys let you define positional(args) and keyword(kwargs) to a custom ConfigTreeView prototype. This works in the same fashion as passing any positional([]) and keyword({}) args to an object in python. A common use/need for this arises when you need your custom treeview to know the index of an attribute that was defined in the config file. The config file makes this easy with the use of the *$index* dotted-key string.

An example of an $index key: *$index.cell_bg_set* Supplying the above $index key in either args or kwargs will set that parameter as the index of 'cell_bg_set'(as defined in *index_names*) in the data model. Values supplied to *args* and *kwargs* are passed through the 'handle_args' function in the ConfigTreeView which should be overridden when you create your own custom ConfigTreeView.

The *bg*, *bg-even*, and *bg-odd* keys are style options that I've added that allow you to change the background colors of a TreeView. *bg* by itself changes the background to a single color...color can either be a gtk-accepted string name(ex: "blue") or it can be a hexidecimal string(ex: "#FF0078"). If you wish to have your ConfigTreeView with two colors, alternating per row, supply the *bg-even* and *bg-odd* colors instead. Note: *bg-even* and *bg-odd* must be supplied together or you'll get errors when you try to apply your config. A GTK Note: These color options change the underlying GtkStyle which is determined by the users theme. You should try to stay away from using this and respect a user's theme choices...but if you're in a controlled environment or it's for personal use, then it's okay to change these styles as you see fit.

The *selection-mode* key allows you to set the mode of the GtkTreeSelection. This is limited to these values: (SELECTION_NONE, SELECTION_SINGLE, SELECTION_BROWSE, SELECTION_MULTIPLE)

The *selection-color* key allows you to change the color of the selection bar in the TreeView. This bar is what highlights a row you've just clicked on. The value for this key follows the same guidelines as *bg*, *bg-even* and *bg-odd* Note: This too should only be used in a controlled environment, because you should respect the user's choice of theme.

The *treemodel* key(optional):

This optional key defines a TreeModel instance that is required to properly manage and store the data so that it can be displayed by your `ConfigTreeView`. When a call to `ConfigTreeView.get_treemodel()` is made, the information for the TreeModel to create is grabbed from this key's structure:

```
{
        #Config stuff here...
        "treemodel": {
                "module": "chronicle.gui.tools.image_loader",
                "class": "ImageStore",
                "args": ["$index.market.pixbuf"],
                "kwargs":{},
        }
        #More config stuff here...
}
```

- The *"module"* key is a string that points to the location of the module that you need to import in order to create an instance of the custom TreeModel you need to use.
- The *"class"* key is a string that is the name of the custom TreeModel that will be instantiated. Note: This class must be an instance of a *GtkTreeModel* or the instantiation will fail and fallback to a *GtkListStore*.
- The *"args"* key is a list of all the positional arguments you want to send to the TreeModel's *_handle_args()* method. This method should be present in your TreeModel custom implementation if you need to pass any arguments to it that will be handled before data rows are appended to the model.
- The *"kwargs"* key is a dict of keyword arguments you to send to the TreeModel's *_handle_args()* method. The same rules apply to this key as to the *"args"* key.

It is important to note that if this key is not supplied, or there is any kind of error in importing, initializing, or running the custom implementation defined in this structure–the ConfigTreeView will fallback and initialize a *GtkListStore* with the proper *types* and return that instead when *get_treemodel()* is called.

The *index_names* key(required):

This required key defines how your data structure will look, as well as defining the types for each member of a row of data. It essentially defines what attributes a row of data needs, and the types for each attribute. An example:

```
{
  "index_names":{
          "record_open_bool": bool,
          "order_status":{
                  "markup": str,
          },
          "column_one":[
                  {"markup": str, "cell-background":str},
                  {"pixbuf": "gtk.gdk.Pixbuf"}
```

```
            ]
    }
}
```

The top level keys within the *index_names* dict are names that you give to columns, or names that you give to a variable. And the values at these keys result in a type for that value later on.

In the above example the "record_open_bool" key is an example of how you'd define a variable. A 'variable' in the config file is a value that you want an index for in the data model, but it is used by more than one CellRenderer. You'll see how this is applied when the *columns* structure is detailed later on.

The *order_status* key in the above example is for a column named *order_status* that has a single CellRenderer that wishes to create an index for "markup" with a type of 'str'.

The *column_one* key in the above example is for a column named *column_one* that has two CellRenderers. The first CellRenderer defines a "markup" property of type str and a "cell-background" property of type str. And the second CellRenderer defines a "pixbuf" property of type "gtk.gdk.Pixbuf".

Note: when creating *$index* args to pass to "treeview.args/kwargs" you can define any value within the *index_names* dict by using dotted-key-notation. Examples:

```
"$index.record_open_bool"
"$index.order_status.markup"
"$index.column_one.0.markup"
"$index.column_one.1.pixbuf"
```

The *column_order* key(required):

This required key defines the order you want the columns to be appended to the ConfigTreeView. Example:

```
"column_order": ["order_status", "column_one"]
```

The column names are the same as the columns that you define in *index_names* and also *columns*.

The *macros* key(optional):

This key lets you define a set of properties that you wish to use multiple times throughout the config file. This feature is added only as a convenience to make the config structure cleaner looking by removing some redundancies. Example:

```
{
        "macros":{
                "cell-text-default":{
                        "font": "Lucida Sans 8",
                        "foreground": "green",
                }
        }
}
```

In the above example, a macro named "cell-text-default" was defined that sets properties "font" and "foreground". These properties must be valid GtkProperties for whatever widget you end up defining them for(TreeViewColumn or CellRenderer)

In the next section, *columns*, you'll see how to set a macro that's been defined.

The *columns* key(required):

This required key lets you define the columns, by name, that you want to create for this ConfigTreeView. The name must be the same name as defined in *index_names*. Example:

```
"columns":{
        "column_one":{
        },
        "order_status"{
        }
}
```

Example of "column_one" column:

```
"column_one":{
        "header":{
                "title": "Column1",
                "module": None,
                "class": "Button",
                "args": [],
                "kwargs": {},
        },
        "macros": ["cell-text-default"],
        "properties"{
                "resizable": True,
                "visible": True,
                "max-width": 100,
        },
        "renderers":[]
}
```

The *header* key is where you define anything about this column's header. If you just want text in the header, then supply the *title* key. You may wish to put a widget up there which you can do by supplying the 'module', 'class', 'args', and 'kwargs' keys for the class you want. Note: if module isn't supplied, it's defaulted to the gtk module. And if class isn't supplied, it's defaulted to gtk.Label.

The *macros* key is where you set the macros that you already defined in the *macros* top-level-key. This is simply done by supplying the name of the macro(s) you wish to use in a list as is demonstrated in the above example.

The *properties* key is where you define any properties that you want to set for the particular column. This follows the same guidelines as the *properties* key in the *treeview* top-level key. TreeViewColumn properties: http://www.pygtk.org/docs/pygtk/class-gtktreeviewcolumn.html

Defining Renderers:

The *renderers* key is where you define what CellRenderers will go into a given column and also what properties and indices the renderer will have. Example:

```
"column_one":{
        "renderers":[
                {
                        "pack": "pack_start",
                        "expand": "True",
                        "module": None,
                        "class": "CellRendererText",
                        "args": [],
```

```
                        "kwargs": {},
                        "properties":{
                                "height": 25,
                                "xpad": 5,
                                "font": "Times New Roman 13"
                        },
                        "indices": {
                                "markup": True,
                                "cell-background-set": "record_open_bool"
                        }
                }
                {
                        "class": "CellRendererPixbuf",
                        "indices": {
                                "pixbuf": True,
                                "cell-background-set": "record_open_bool"
                        }
                }
        ]
}
```

If a column has a single CellRenderer, then you define the *renderers* key as a dict({}), but if a column has multiple CellRenderers, you define the *renderers* key as a list([]) containing dicts where each dict is a CellRenderer

From the above example you see a CellRenderer is defined as: The *pack* key defines how the CellRenderer will be added to the TreeViewColumn. Accepted values are: "pack_start" and "pack_end". If this key isn't supplied, it defaults to "pack_start".

The *expand* key defines whether or not the CellRenderer will expand to the size the TreeViewColumn supplies. True or False. If key isn't supplied, it defaults to True.

The *module*, *class*, *args*, and *kwargs* keys are used to supply a custom CellRenderer. If these keys aren't supplied, it's defaulted to a gtk.CellRendererText

The *properties* key is where you define any properties that you want to set specifically for this CellRenderer. This follows the same guidelines as the *properties* key in the *treeview* top-level key. CellRendererProperties: http://www.pygtk.org/pygtk2tutorial/sec-CellRenderers.html

The *indices* key is where you define what CellRenderer properties you want controlled by the data model. You already defined the types and size of the structure, this is where you request an index in the data model for the given property. These properties are also the same properties that you can define in the *properties* key, but when they're defined as an index, the value of the property is defined by the data model for each row.

In the above example, the first CellRenderer dict defines two properties: "markup" and "cell-background-set". When you're just requesting a new index, a value of True is passed...but when you're setting that value to another value(granted the value is defined in *index_names*) then you're simply pointing that property to an index that is shared by multiple renderers.

CHAPTER 2

API Documentation:

configtreeview Module

configtreeview

Subpackages

configtreeview.tools Package

dataformatter Module

CHAPTER 3

Examples:

Examples

This section details everything you need to get started using `ConfigTreeView` s. This includes an example of how to use the `ConfigTreeView`, how to create the *config* structure and also how to create a custom wrapper to extend its uses to fit your needs and finally how to use the *DataFormatter* to get your data displayed correctly in the `ConfigTreeView`.

How To Use a *ConfigTreeView*

To use a *ConfigTreeView* is simple. In your python script there are only a couple steps you need to follow:

1. Initialize the *ConfigTreeView* with a config structure
2. Apply the configuration to finish initializing the *ConfigTreeView* components

Example:

```
#Import the package
from configtreeview import ConfigTreeView
#Create the ConfigTreeView
my_treeview = ConfigTreeView(config) #where `config` is a python dict config␣
↪structure you already defined
#Apply the configuration to the TreeView to finish initialization
my_treeview._apply_config()
```

As you can see from the example, the API to use a ConfigTreeView is incredibly simple. This is possible because the bulk of the work is done in how you define and build the *config* structure.

How To Create/Build a *config* Structure

The config structure itself is just a python *dict* instance that contains the information necessary to build all of the interface components for the *ConfigTreeView*. For more information on what a *config* structure should look like, refer

to *HOW TO: Config File*

As an example let's say we want a very simple treeview that contains two columns that each have a title in their header, and contain a single string as the data source for their respective *GtkCellRenderers*. The config structure for such a case could look like this:

```
{
        "index_names": {
                'column_1': {'markup': str},
                'column_2': {'markup': str}
        },
        "column_order": ["column_1", "column_2"],
        "columns":{
                "column_1":{
                        "header":{
                                "title": "Column1"
                        },
                        "renderers": {
                                        "indices":{"markup": True}
                        }
                },
                "column_2":{
                        "header":{
                                "title": "Column2"
                        },
                        "renderers": {
                                        "indices":{"markup": True}
                        }
                }
        }
}
```

A Custom *ConfigTreeView* implmentation

Sometimes it is necessary to create your own implementation of a `ConfigTreeView`. This section describes the how? and the why?

Why?

There are many situations in your interface where you may need to subclass the `ConfigTreeView` so as to handle something extra that it doesn't cover. One common situation that has come up in my own experiences has been with needing to know one of the indexes represented in the data model for some dynamic use throughout your application. For example: you would like to change the background color of a cell in the treeview based on a user clicking on it and to do this you need to know which index in a TreeModel row you're keeping that property. Because of the way the data model is built for use in the `ConfigTreeView` with the actual indices hidden from the developer, this is not immediately an easy task.

How?

But, fear not, the `ConfigTreeView` supplies an easy way to get around this. To achieve this, you need to do a few things:

1. Pass the index that you want(in dotted-key notation) to the *args'(or 'kwargs*) of the *treeview* dict within your config structure.

2. Create a custom wrapper class that subclasses *ConfigTreeView*
3. Make sure you override the *_handle_args* function of *ConfigTreeView* that will be used to assign the args you specified in the config structure to an attribute of your custom *ConfigTreeView* that you can use.

Example: building on from the above sample of a config structure...let's add a variable to our config that will be used to change the background of a cellrenderer:

```
{
        "treeview": { #Added a 'treeview' key to supply custom args to our␣
↪treeview
                "args": ['$index.bg_color'],#Pass the 'bg_color' index to␣
↪our custom treeview
        },
        "index_names": {
                'bg_color': str,#Define the 'bg_color' variable--we'll make␣
↪it a variable if we plan on using the same index for multiple renderers
                'column_1': {'markup': str},
                'column_2': {'markup': str}
        },
        "column_order": ["column_1", "column_2"],
        "columns":{
                "column_1":{
                        "header":{
                                "title": "Column1"
                        },
                        "renderers": {
                                        "indices":{"markup": True}
                        }
                },
                "column_2":{
                        "header":{
                                "title": "Column2"
                        },
                        "renderers": {
                                        "indices":{"markup": True}
                        }
                }
        }
}
```

And then a corresponding custom treeview implementation would look something like this:

```
class CustomConfigTV(ConfigTreeView):
        '''
        An example of a custom ConfigTreeView wrapper
        used to get indices that were defined in
        the config structure
        '''

        def __init__(self, *args):
                ConfigTreeView.__init__(self, *args)
                #Any post-initializing stuff here
                #This is the place to do stuff *BEFORE* the
                #columns, renderers, and properties are set
                self.background_idx = None

        def _handle_args(self, background_idx):
                '''
```

```
                Override this function and set your custom args
                '''
                self.background_idx = background_idx
```

And now your treeview has an attribute *background_idx* that will contain the index at which the 'bg_color' property you defined in the config will be formatted to.

Using DataFormatter to create rows

Now that you've created a `ConfigTreeView` and initialized it with a config structure, you're ready to start giving it data and using it! In order to get the data properly displayed you need to do a few things:

1. Make sure you understand the *ConfigTreeView data model* and use it to give properly constructed data sets
2. Create a *DataFormatter* object, initializing it with your `ConfigTreeView` 's *index_map*, and *types* structures.
3. Create a *GtkTreeModel* to supply your TreeView with data
4. Using the *DataFormatter* to yield formatted rows that can then be appended to your *GtkTreeModel*

Example(using the config that was defined above):

```
from configtreeview.tools import DataFormatter

data = [ #The DataFormatter is expecting a list of dicts...each dict is a␣
↪row following the ConfigTreeView Data Model
        {'column_1': {'markup': 'Some data here'}, 'column_2': {'markup':
↪'Column 2 data here'}},
        {'column_1': {'markup': 'More data col1'}, 'column_2': {'markup':
↪'Column 2 data here'}},
        {'column_1': {'markup': 'Even more data here'}, 'column_2': {'markup
↪': 'Column 2 data here'}},
]

#Create the DataFormatter
data_formatter = DataFormatter(my_treeview.index_map, mytreeview.types)

#Get a treemodel
liststore = my_treeview.get_treemodel() #Use this function to have your␣
↪treeview build you a proper GtkTreeModel instance
my_treeview.set_model(liststore)

#Format the data rows for liststore and append them to it
for row in data_formatter.get_rows(data):
        liststore.append(row)
```

And now the data should be displayed by your treeview!

CHAPTER 4

Indices and tables

- genindex
- modindex
- search