ASCII Designer Documentation

Release 0.3.1

Johannes Löhnert

Sep 28, 2020

Contents

CHAPTER 1

ASCII Designer Manual

1.1 What is this?

A library that:

- creates GUI from ASCII-art (with well-defined syntax)
- maps widgets to virtual class attributes
- relieves you from the boring parts of Form building while leaving you in control.

The workhorse class is the *AutoFrame*:

```
from ascii_designer import AutoFrame
```

The created widgets are **"raw", native widgets**. You do not get wrappers; instead, the library focuses on specific tasks - building the layout, event-/value binding - and lets you do everything else with the API you know and (maybe) love.

1.2 AutoFrame overview

AutoFrame is used by subclassing it. Then, define the special attribute `f_body` to define the design:

```
class MyForm(AutoFrame):
    f_body = '''
        |            |
         Hello World!
          [Close]
    '''
```

That's it: A working form. Show it by calling *f_show()*. If necessary, it will set up an application object and main loop for you; the boilerplate code reduces to:

```
if __name__ == '__main__':
    MyForm().f_show()
```

You can set the `f_title` attribute for a custom window title. Otherwise, your class named is turned into a title by space-separating on uppercase characters.

If you like menus, `f_menu` can be used for concise definition of menu structures.

Finally, there is the *`f_build()`* method, which does the actual form generation. This is the method to override for custom building and initialization code.

1.3 Grid slicing, stretching and anchors

ASCII designer generates grid layouts. The first step of processing `f_body` is to cut it up into text "cells". Each line of the `f_body` string is converted into one grid layout row.

Before processing, leading and trailing whitespace lines are cropped. Also, common leading whitespace is removed.

1.3.1 Slicing

The first line is used to define the split points by means of the pipe character (`|`). The lines below are split exactly underneath the pipe signs, IF the respective text-column is either space or pipe character. If, on the other hand, any other character is present in that line and text-column, a column-spanning cell is created, containing the joint text of both cells.

If you want to join but have a space character at this point, you can use the tilde `~` character instead of the space. It will be converted to space in the subsequent processing.

Row-spans are created by prepending the cell content with a brace `{` character. No matching close-brace is needed. The brace characters must be aligned, i.e. exactly above each other.

1.3.2 Stretch

By default, column widths will "shrinkwrap" the content. To make a column stretchable, insert one or more minus `-` signs in the first line between the pipe chars:

```
|    -      |             |
 stretches   fixed width
```

If you want it nice-looking, you can make a double arrow like so: `<-->`; however to the program only the minus characters count.

If you define multiple stretchable columns, the stretch proportion of each column is equal to the number of minus chars above.

Row-stretch is defined similarly. You need to create a special "first text-column" by having a pipe char before any content underneath:

```
|              |               |
 <- special text-column
 column 1       column 2
```

In this text-column, put a capital `I` before rows that you want to stretch. Stretch proportion is equal for all stretchable rows. Use row-span to have some widgets stretch more than others vertically.

1.3.3 Anchoring

Anchoring refers to the positioning and stretching of the widget *within* its grid cell.

Horizontal anchoring of widgets within the grid cell is controlled by whether the text is space-padded at the beginning and/or end of its text cell:

- No space at beginning nor end makes the widget full-width.
- Space at only beginning gives right-, at end gives left-align.
- Space at both begin and end gives center alignment.

In graphical form:

```
|                 |
  Alignment:
 [Fill           ]
 [Left]          ~
           [Right]
    [Center]     ~
  [also center ] |
```

Note how tilde character is used as space substitute. This is because trailing space is badly visible, and also removed by some text editors automatically. The last row shows another possibility by explicitly putting a pipe sign at the end.

Vertical anchoring is not controllable. It defaults to "fill", which is the right thing most of the time. If not, you can use toolkit-native methods to change the anchoring afterwards.

1.4 Widget specification

To create a:	Use the syntax:
Label	`blah blah` (just write plain text), `label_id:  Text` or `.Text`
Button	`[ ]` or `[Text]` or `[control_id:  Text]`. (From here on simplified as `id_and_text`).
Text field	`[id_and_text_]` (single-line) or `[id_and_text__]` (multi-line)
Dropdown Chooser	`[id_and_text v]` or `[id_and_text (choice1, choice2) v]`
Combobox	`[id_and_text_ v]` or `[id_and_text_ (choice1, choice2) v]`
Checkbox	`[ ] id_and_text` or `[x] id_and_text`
Radio button	`( ) id_and_text` or `(x) id_and_text`
Slider (horizontal)	`[id:  0 -+- 100]`
List/Tree view (only in Tk for now)	`[= id_and_text]` or `[= id_and_text (Column1, Column2)]`
Placeholder (empty or framed box)	`<name>` for empty box; `<name:Text>` for framed box

1.4.1 Control ID

Each control gets an identifier which is generated as follows:

- If a control id is explicitly given, it has of course precedence.
- Otherwise, the control Text is converted to an identifier by
  - replacing space with underscore
  - lower-casing
  - removing all characters not in (a-z, 0-9, _)
  - prepending `x` if the result starts with a number.
  - Special-Case: Labels get `label_` prepended.
- If that yields no ID (e.g. Text is empty), the ID of a preceding Label (without `label_` prefix) is used. This requires the label to be *left* of the control in question.
- If that fails as well, an ID of the form `x1`, `x2`, ... is assigned.

Examples:

- `[ Hello ]` gives id `hello`
- `[ Hello World! ]` gives id `hello_world`
- `Hello World: | [ ]` gives a label with id `label_hello_world` and a button with id `hello_world`
- `[ $%&§§% ]` gives a button with id `x1` (assuming this is the first control withoud id).

The control id can be used to get/set the control value or the control object from the form - see below.

1.4.2 Notes about specific widgets

Dropdown and **combobox** without values can be populated after creation.

All **radio buttons** on one form are grouped together. For multiple radio groups, create individiual AutoFrames for the group, and embed them in a box.

Slider: only supported with horizontal orientation. For a vertical slider, change orientation afterwards; or use a placeholder box and create it yourself.

Listview: The first column will have the text as heading. The subsequent columns have the given column headings. If Text is empty (or only id given), only the named columns are there. This makes a difference when using value-binding (see below).

1.5 Value and event binding

1.5.1 Control objects

Usually you will access your controls from methods in your *`AutoFrame`* subclass. So let us assume that your `AutoFrame` variable is called `self`.

Then, access the generated controls by using `self["control_id"]` or `self.f_controls["control_id"]`. The result is a toolkit-native widget, i.e. a `QWidget` subclass in Qt case, a `tkinter` widget in Tk case.

For Tk widgets, if there is an associated Variable object (`StringVar` or similar), you can find it as `self["control_id"].variable` attribute on the control.

1.5.2 Event binding

If you define a method named after a control-id, it will be automatically called ("bound", "connected") as follows:

- Button: When user clicks the button; without arguments (except for `self`).
- Any other widget type: When the value changes; with one argument, being the new value.

Example:

```
class EventDemo(AutoFrame):
    f_body = '''
        |               |
         [ My Button   ]
         [ Text field_ ]
    '''
    def my_button(self):
        print('My Button was clicked')

    def text_field(self, val):
        print('Text "%s" was entered'%val)
```

In case of the ListView, the method is called on selection (focus) of a row.

As second option, you can name the method `on_<control-id>` (e.g.: `on_text_field`). Thus the handler can easily coexist with the virtual value attribute (read on).

1.5.3 Virtual value attribute

If the control is not bound to a function, you can access the value of the control by using it like a class attribute:

```
class AttributeDemo(AutoFrame):
    f_body = '''
        |               |
         [ Text field_ ]
    '''
    def some_function(self):
        x = self.text_field
        self.text_field = 'new_text'
```

For label and button, the value is the text of the control.

Boxes are a bit special. An empty box's value is the box widget itself. A framed box contains an empty box, which is returned as value.

You can set the virtual attribute to another (any) widget the toolkit understands. In this case, the original box is destroyed, and the new "value" takes its place. For a framed box, the inner empty box is replaced. So you can use the box as a placeholder for a custom widget (say, a graph) that you generate yourself.

Note: The new widget must have the same parent as the box you replace.

A second possibility is to use the box as parent for one or more widgets that you add later. For instance, you can render another AutoFrame into the box. (see under Extending).

1.5.4 Value of List / Tree View

Note: Lists and tree views are considerably more complex than the other widgets. I am still experimenting with how to make handling as convenient as possible. Be prepared for changes here if you update.

The general picture is this: The Listview has a value, which on the python side looks mostly like a list. You can slice it, insert/remove items and so on.

Inserted items are displayed in the list view in textual form. The value list is attached to the actual list view. I.e. if you update the list, the changes immediately reflect in the ListView widget.

The value list or its items can become detached if you replace the list or pop nodes of it. You can still use it like a normal python object, but it will not have an onscreen representation anymore.

The *sources* method of the list can be used to configure how values are read from the given objects into the predefined columns. By default we look for attributes matching the column names. If you have a first column (defined via the "Text", not the "Columns" list in parens), it gets the object's string representation.

That means that the simplemost way of using the List is this:

```
class SimpleList(AutoFrame):
    f_body = '''
        |
         [= Some Items]
    '''
    def f_build(self, parent, body):
        super().f_build(parent, body)
        # populate the list
        self.some_items = ['First', 'Second', 'Fifth']
```

A more complex example to showcase how additional columns work:

```
# RankRow is a stand-in for a "real" class.
RankRow = namedtuple('RankRow', 'name points rank')

class TreeDemo(AutoFrame):
    f_body = '''
    |              <->                |
    I[= Players (,Name, Points, Rank)]
    '''
    def f_build(self, parent, body):
        super().f_build(parent, body)
        self.players = [
            RankRow('CaptainJack', 9010, 1),
            RankRow('MasterOfDisaster', 3010, 2),
            RankRow('LittleDuck', 12, 3),
        ]
        # Replacing items triggers updating of the displayed data
        self.players[2] = RankRow('BigDuck', 24, 3)
        # change the data binding:
        self.players.sources(
                lambda obj: 'ItsLikeMagic',  # unnamed arg: sets the default text␣
↪(first column)
                name=['foo'], points=['bar'], # use __getitem__ for those
                # custom callback
                rank=lambda obj: obj['baz'],
```

(continues on next page)

(continued from previous page)

```
        )
        self.players.append({'foo': 'Last', 'bar': -1, 'baz': 4})
```

When working with the list, keep in mind that it **can be changed by user interaction** (like any other widget's value). Currently the only possible change is to re-sort the list, but more (edit, add, remove items) might come.

Note: Currently Tk and Qt toolkit behave notably different concerning lists. Tk retrieves the "source" values once to build all the list items. Meaning that changes in the underlying items do not reflect in the list unless explicitly updated.

Qt on the other hand queries the items permanently (e.g. on mouse-over). This means that changes are immediately visible onscreen, but that you should not do complicated calculations or I/O to retrieve column values.

Trees are created by using the *ObsList.children_source* method, which works similar to *sources*. Here you can define two sources, one for `has_children` (bool) and one for `children` (list).

The tree is lazy-loading, i.e. children are only retrieved when a node is expanded. On repeated expansion, children are reloaded.

`has_children` is queried to determine whether expanders should be drawn on each item. If not given, we assume that each entry might have children, and they all get expanders initially.

The `children` property, if retrieved, is again a special list like the "root" one.

To identify items in the tree, the two methods *ObsList.find* and *ObsList.find_by_toolkit_id* are provided, which yield container list and index given the item or its toolkit-native identifier, respectively.

For Tk, the toolkit-native identifier is the `iid` value of the tree item.

For Qt it is unset; only `parent_toolkit_id` is set to the parent `QModelIndex`. Given a `QModelIndex`, its `internalPointer()` refers to the containing list and `row()` gives the index of the item.

1.6 Menus

Define menus by setting (overriding) the f_menu property of your AutoFrame. An example menu looks like this:

```
f_menu = [
    "File >", ["Open", "Save", "Quit"],
    "Nested >", [
        "Item 1 #C-I",
        "Submenu 1 >", [ "Subitem 1"],
        "Item 2",
    ],
]
```

There are two kinds of menu entries:

- Normal actions are just simple strings. An identifier is created from the text according to the rules above, e.g. `item_1` for the text `"Item 1"`. The `AutoFrame` **must** have a method of that name and without parameters except `self`. It will automatically be bound to the menu entry.
- If on the other hand the text ends with `>`, it defines a submenu. The next list entry is expected to be a nested list defining the submenu. No handler function is bound to the submenu label.

Normal actions can be followed by a shortcut definition introduced by hash sign `#`. It can contain any of `C-`, `A-`, `S-` modifiers followed by a letter.

For common actions like Open/Save or Cut and Paste, Shortcuts are generated automatically. Those are defined in *ToolkitBase.default_shortcuts*. (In *ToolkitQt* this map is overriden to use the `QKeySequence.X` defaults).

The menu is generated by the *AutoFrame.f_build_menu* function, which is called from *f_show*. No menu is built if using *f_build* directly (since you are most likely embedding the frame). If crucial functionality is missing because of this, it is your own fault...

1.7 Extending / integrating

In any real-world scenario, you will hit the limits of this library pretty soon. Usually it boils down to one of the questions:

- How do I use toolkit-native methods on the widgets?
- How can I embed generated controls into a "3rd-party" window?
- How can include "3rd-party" controls in the generated grid?

1.7.1 Toolkit-native methods

Having an AutoFrame `self`, access the toolkit-native controls by using `self["control_id"]` or `self.f_controls["control_id"]`. Do whatever you like with them.

1.7.2 Embedding `AutoFrame` into a 3rd-party host window

The *AutoFrame.f_build* method takes a parent window as argument. You can use this to "render" the AutoFrame into a custom container.

- The container can be any widget taking children. It must be preconfigured to have a grid layout. I.e. for `tk` toolkit, `.pack()` must not have been used; in case of `qt` toolkit, a `QGridLayout` must have been set via `.setLayout()`.
- Already-existing children are ignored and left in place. However, row/column stretching is modified.
- Automatic method / property binding works as usual.

1.7.3 Including 3rd-party controls into an `AutoFrame`

This is what the `<placeholder>` control is for. It creates an empty Frame / Widget / Panel which you can either use as parent, or replace with your own control.

For the former, get the placeholder object (via its value attribute) and use it as parent. You must do the layout yourself.

For the latter, set its virtual value attribute to your widget. This destroys the placeholder. The layout of the placeholder (Grid position and stretching) is copied onto the new widget.

1.7.4 Nesting `AutoFrame`

Combining both methods, you can also embed one AutoFrame into another. The following example showcases everything:

```
class Host(AutoFrame):
    f_body = '''
        |
         <placeholder>
    '''
    def f_build(self, parent, body=None):
        super().f_build(parent, body)
        # self.placeholder.setLayout(QGridLayout()) # only for Qt

        # create instance
        af_embedded = Embedded()
        # render widgets as children of self.placeholder
        af_embedded.f_build(parent=self.placeholder)
        # store away for later use
        self._embedded = af_embedded

class Embedded(AutoFrame):
    f_body = '''
        |
         <another placeholder>
    '''
    def f_build(self, parent, body=None):
        super().f_build(parent, body)
        parent = self.another_placeholder.master
        self.another_placeholder = tk.Button(parent, text='3rd-party control')
```

CHAPTER 2

API documentation (autogenerated)

2.1 *ascii_designer* package

From `ascii_designer` you can directly import:

- The *`AutoFrame`* class
- The *`set_toolkit`* function.

2.2 *ascii_designer.autoframe* module

Warning: module 'ascii_designer.autoframe' undocumented

class ascii_designer.autoframe.**AutoFrame**
class name is converted to title. Override with f_title.

Body definition with f_body.

To create own widgets or customize the autocreated ones, override *`f_build`*.

Get at the created controls using AutoFrame[key].

close(), exit(), quit() provided for convenience.

Functions with same name as a control are autobound to the default handler (click or changed).

Attributes are autobound to the control value (get/set), except if they are explicitly overwritten.

close()
Close the window.

This is also called when the window is closed using the x button. Be sure to call `super().close()` or your window won't close.

exit()

f_add_widgets(*parent*, *sliced_grid=None*, *body=None*, *offset_row=0*, *offset_col=0*, *autoframe=None*)

f_build(*parent*, *body=None*)

f_build_menu(*parent*, *menu=None*)

Builds the menu from the given menu definition.

Menu definition is a list which can (currently) contain actions and submenus.

An Action is simply a string, which is converted to an identifier following the same rules as the other widgets. It triggers the `self.` method named as the identifier. The method must be defined.

A submenu is created by a string ending in ">", followed by an item which is itself a list (the submenu content).

Example

```
>>> menu = [
        'File >', ['Open', 'Save', 'Quit'],
        'Help >', ['About'],
    ]
>>> autoframe.f_build_menu(autoframe.f_controls(''), menu)
```

f_show()

Bring the frame on the screen.

quit()

2.3 *ascii_designer.ascii_slice* module

Functions to slice up a fixed-column-width ASCII grid.

slice_grid splits up lines according to a header row with | separators.

merged_cells iterates over this grid and returns merge areas.

Columns are merged if there is something different from | or space below the separator in the header row.

Rows are merged by prefixing the cells with {. The symbols must be in the same text column.

ascii_designer.ascii_slice.**slice_grid**(*grid_text*)

slice a grid up by the first (nonempty) row.

Before slicing, empty lines before/after are removed, and the text is dedented.

The first row is split by | characters. The first column can contain a | character or not.

Returns a SlicedGrid with Properties:

- column_heads: the split up parts of the first line (not including the separators).
- body_lines: list of following lines; each item is a list of strings, where each string is the grid "cell" including the preceding separator column. I.e. if you join the cell list without separator, you regain the text line.

class ascii_designer.ascii_slice.**SlicedGrid**(*column_heads=NOTHING*, *body_lines=NOTHING*)

Warning: class 'ascii_designer.ascii_slice.SlicedGrid' undocumented

ascii_designer.ascii_slice.**merged_cells**(*sliced_grid*)

Generator: takes the sliced grid, and returns merged cells one by one.

Cells are merged by the following logic:

- If the first character of a (stripped) cell is '{', cells of the following row(s) are merged while they also start with '{' in the same column.
- Then, columns are merged if the following (column's) cell starts neither with space nor with '|'.

Yields MCell instances with:

- row, col: cell position (int, 0-based)
- rowspan, colspan: spanned rows/cols, at least 1
- text: merged area text, as sliced out from the text editor; not including the leading '{'; "ragged" linebreaks retained.

Iteration order is row-wise.

Merge areas must not overlap. (However this should rarely happen on accident).

Note: If you need two row-merge ranges above each other, indent the '{' differently.

class ascii_designer.ascii_slice.**MCell**(*row*, *col*, *text=''*, *rowspan=1*, *colspan=1*)

Warning: class 'ascii_designer.ascii_slice.MCell' undocumented

2.4 *ascii_designer.toolkit* module

Warning: module 'ascii_designer.toolkit' undocumented

ascii_designer.toolkit.**set_toolkit**(*toolkit_name*)

Warning: function 'ascii_designer.toolkit.set_toolkit' undocumented

ascii_designer.toolkit.**get_toolkit**()

Warning: function 'ascii_designer.toolkit.get_toolkit' undocumented

class ascii_designer.toolkit.**ToolkitBase**

Warning: class 'ascii_designer.toolkit.ToolkitBase' undocumented

anchor (*widget*, *left=True*, *right=True*, *top=True*, *bottom=True*)
anchor the widget. Depending on the anchors, widget will be left-, right-, center-aligned or stretched.

box (*parent*, *id=None*, *text=''*, *given_id=''*)
An empty panel (frame, widget, however you call it) or group box that you can fill with own widgets.

`given_id` is the user-given id value, as opposed to `id` (the autogenerated one). A Group box is created if text AND given_id are set.

The virtual attribute value is the panel itself, or in case of groupbox the contained panel.

button (*parent*, *id=None*, *text=''*)

checkbox (*parent*, *id=None*, *text=''*, *checked=None*)
Checkbox

close (*frame*)
close the frame

col_stretch (*container*, *col*, *proportion*)
set the given col to stretch according to the proportion.

combo (*parent*, *id=None*, *text=''*, *values=None*)
combo box; values is the raw string between the parens. Free-text allowed.

connect (*widget*, *function*)
bind the widget's default event to function.

Default event is:

- click() for a button
- **value_changed(new_value) for value-type controls;** usually fired after focus-lost or Return-press.

default_shortcuts = {'copy': 'C-C', 'cut': 'C-X', 'find': 'C-F', 'new': 'C-N', 'op

dropdown (*parent*, *id=None*, *text=''*, *values=None*)
dropdown box; values is the raw string between the parens. Only preset choices allowed.

getval (*widget*)
get python-type value from widget.

grammar = [('box', '\\<(?:\\s*(?P<id>[a-zA-Z0-9_]+)\\s*\\:)?\\s*(?P<text>[^(]*?)?\\s*

label (*parent*, *id=None*, *label_id=None*, *text=''*)

menu_command (*parent*, *id*, *text*, *shortcut*, *handler*)
Append command labeled `text` to menu `parent`.

Handler: `func() -> None`, is immediately connected.

`shortcut` follows the syntax `(modifier)-(key)`, where `modifier` is one or more of `C`, `S`, `A` for Ctrl, Shift, Alt respectively.

menu_grammar = [('sub', '(?:\\s*(?P<id>[a-zA-Z0-9_]+)\\s*\\:)?\\s*(?P<text>[^(]*?)?\\s

menu_root (*parent*)
Create menu object and set as parent's menu.

menu_sub (*parent*, *id*, *text*)
Append submenu labeled `text` to menu `parent`.

multiline(*parent*, *id=None*, *text=''*)
multiline text entry box

option(*parent*, *id=None*, *text=''*, *checked=None*)
Option button. Prefix 'O' for unchecked, '0' for checked.

parse(*parent*, *text*)
Returns the widget id and widget generated from the textual definition.

Autogenerates id:

- If given, use it
- else, try to use text (extract all `a-z0-9_` chars)
- else, use 'x123' with 123 being a globally unique number

For label type, id handling is special:

- The label's id will be `"label_"` + id
- The id will be remembered and used on the next widget, if it has no id.

If nothing matched, return None, None.

parse_menu(*parent*, *menudef*, *handlers*)
Parse menu definition list and attach to the handlers

place(*widget*, *row=0*, *col=0*, *rowspan=1*, *colspan=1*)
place widget

root(*title='Window'*, *on_close=None*)
make a root (window) widget. Optionally you can give a close handler.

row_stretch(*container*, *row*, *proportion*)
set the given row to stretch according to the proportion.

setval(*widget*, *value*)
update the widget from given python-type value.

value-setting must not interfere with, i.e. not happen when the user is editing the widget.

show(*frame*)
do what is necessary to make frame appear onscreen.

This should start the event loop if necessary.

slider(*parent*, *id=None*, *min=None*, *max=None*)
slider, integer values, from min to max

textbox(*parent*, *id=None*, *text=''*)
single-line text entry box

treelist(*parent*, *id=None*, *text=''*, *columns=None*)
treeview (also usable as plain list)

`ascii_designer.toolkit.`**auto_id**(*id*, *text=None*, *last_label_id=''*)
for missing id, calculate one from text.

2.5 *ascii_designer.toolkit_tk* module

This is a construction site...

class ascii_designer.toolkit_tk.**ToolkitTk**(**args*, ***kwargs*)
Builds Tk widgets.

Returns the raw Tk widget (no wrapper).

For each widget, a Tk Variable is generated. It is stored in `<widget>.variable` (attached as additional property).

If you create Radio buttons, they will all share the same variable.

The multiline widget is a tkinter.scrolledtext.ScrolledText and has no variable.

The ComboBox / Dropdown box is taken from ttk.

Box variable (placeholder): If you replace the box by setting its virtual attribute, the replacement widget must have the same master as the box: in case of normal box the frame root, in case of group box the group box. Recommendation: `new_widget = tk.Something(master=autoframe.the_box.master)`

anchor(*widget*, *left=True*, *right=True*, *top=True*, *bottom=True*)
anchor the widget. Depending on the anchors, widget will be left-, right-, center-aligned or stretched.

box(*parent*, *id=None*, *text=''*, *given_id=''*)
An empty panel (frame, widget, however you call it) or group box that you can fill with own widgets.

`given_id` is the user-given id value, as opposed to `id` (the autogenerated one). A Group box is created if text AND given_id are set.

The virtual attribute value is the panel itself, or in case of groupbox the contained panel.

button(*parent*, *id=None*, *text=''*)

checkbox(*parent*, *id=None*, *text=''*, *checked=None*)
Checkbox

close(*frame*)
close the frame

col_stretch(*container*, *col*, *proportion*)
set the given col to stretch according to the proportion.

combo(*parent*, *id=None*, *text=''*, *values=None*)
combo box; values is the raw string between the parens. Free-text allowed.

connect(*widget*, *function*)
bind the widget's default event to function.

Default event is:

- click() for a button
- **value_changed(new_value) for value-type controls;** usually fired after focus-lost or Return-press.

dropdown(*parent*, *id=None*, *text=''*, *values=None*)
dropdown box; values is the raw string between the parens. Only preset choices allowed.

getval(*widget*)
get python-type value from widget.

label(*parent*, *id=None*, *label_id=None*, *text=''*)

menu_command(*parent*, *id*, *text*, *shortcut*, *handler*)
Append command labeled `text` to menu `parent`.

Handler: `func() -> None`, is immediately connected.

menu_root (*parent*)
Create menu object and set as parent's menu.

menu_sub (*parent*, *id*, *text*)
Append submenu labeled `text` to menu `parent`.

multiline (*parent*, *id=None*, *text=''*)
multiline text entry box

option (*parent*, *id=None*, *text=''*, *checked=None*)
Option button. Prefix 'O' for unchecked, '0' for checked.

place (*widget*, *row=0*, *col=0*, *rowspan=1*, *colspan=1*)
place widget

root (*title='Window'*, *on_close=None*)
make a root (window) widget

row_stretch (*container*, *row*, *proportion*)
set the given row to stretch according to the proportion.

setval (*widget*, *value*)
update the widget from given python-type value.

value-setting must not interfere with, i.e. not happen when the user is editing the widget.

show (*frame*)
do what is necessary to make frame appear onscreen.

slider (*parent*, *id=None*, *min=None*, *max=None*)
slider, integer values, from min to max

textbox (*parent*, *id=None*, *text=''*)
single-line text entry box

treelist (*parent*, *id=None*, *text=''*, *columns=None*)
treeview (also usable as plain list)

Implementation note: Uses a ttk.TreeView, and wraps it into a frame together with a vertical scrollbar. For correct placement, the .place, .grid, .pack methods of the returned tv are replaced by that of the frame.

Returns the treeview widget (within the frame).

2.6 *ascii_designer.toolkit_qt* module

ToolkitQt-specific notes:

- Alignment / Stretch not 100% reliable so far, if using row/col-span.
- Tree / List widget not available so far
- **closing of form with X button cannot be stopped in the default handler. If** you need to do this, replace (root).closeEvent function.

class ascii_designer.toolkit_qt.**ToolkitQt**(***kwargs*)

> **Warning:** class 'ascii_designer.toolkit_qt.ToolkitQt' undocumented

anchor(*widget*, *left=True*, *right=True*, *top=True*, *bottom=True*)
anchor the widget. Depending on the anchors, widget will be left-, right-, center-aligned or stretched.

box(*parent*, *id=None*, *text=''*, *given_id=''*)
An empty panel (frame, widget, however you call it) or group box that you can fill with own widgets.

`given_id` is the user-given id value, as opposed to `id` (the autogenerated one). A Group box is created if text AND given_id are set.

The virtual attribute value is the panel itself, or in case of groupbox the contained panel.

button(*parent*, *id=None*, *text=''*)

checkbox(*parent*, *id=None*, *text=''*, *checked=None*)
Checkbox

close(*frame*)
close the frame

col_stretch(*container*, *col*, *proportion*)
set the given col to stretch according to the proportion.

combo(*parent*, *id=None*, *text=''*, *values=None*)
dropdown with editable values.

connect(*widget*, *function*)
bind the widget's default event to function.

Default event is:

- click() for a button
- **value_changed(new_value) for value-type controls;** usually fired after focus-lost or Return-press.

default_shortcuts = {'copy': <DeepFakeModule object>, 'cut': <DeepFakeModule object>

dropdown(*parent*, *id=None*, *text=''*, *values=None*)
dropdown box; values is the raw string between the parens. Only preset choices allowed.

getval(*widget*)
get python-type value from widget.

label(*parent*, *id=None*, *label_id=None*, *text=''*)

menu_command(*parent*, *id*, *text*, *shortcut*, *handler*)
Append command labeled `text` to menu `parent`.

Handler: `func() -> None`, is immediately connected.

menu_root(*parent*)
Create menu object and set as parent's menu.

menu_sub(*parent*, *id*, *text*)
Append submenu labeled `text` to menu `parent`.

multiline(*parent*, *id=None*, *text=''*)
multi-line text entry box

option(*parent*, *id=None*, *text=''*, *checked=None*)
Option button. Prefix 'O' for unchecked, '0' for checked.

place(*widget*, *row=0*, *col=0*, *rowspan=1*, *colspan=1*)
place widget

root (*title='Window'*, *on_close=None*)
make a root (window) widget

row_stretch (*container*, *row*, *proportion*)
set the given row to stretch according to the proportion.

setval (*widget*, *value*)
update the widget from given python-type value.

value-setting must not interfere with, i.e. not happen when the user is editing the widget.

show (*frame*)
do what is necessary to make frame appear onscreen.

slider (*parent*, *id=None*, *min=None*, *max=None*)
slider, integer values, from min to max

textbox (*parent*, *id=None*, *text=''*)
single-line text entry box

treelist (*parent*, *id=None*, *text=''*, *columns=None*)
treeview (also usable as plain list)

Qt notes: The model does no caching on its own, but retrieves item data all the time. I.e. if your columns are costly to calculate, roll your own caching please.

2.7 *ascii_designer.list_model* module

Pythonic list and tree classes that can be used in a GUI.

The general concept for Lists is this:

- List is wrapped into an *ObsList* (by *ToolkitBase.setval*)
- The "code-side" of the ObsList quacks (mostly) like a regular list.
- **The "gui-side" of the ObsList**
  - provides COLUMNS (key-value items) dynamically retrieved from each list item
  - remembers column and order to be used when sorting
  - has a notion of "selection" (forwarded to a callback)
  - provides event callbacks for insert, replace, remove, sort.
- Internally, the *ListMeta* class holds the state data for those functions.

class ascii_designer.list_model.**ObsList** (*iterable=None*, *keys=None*, *meta=None*, *toolkit_parent_id=None*)
Base class for treelist values.

Behaves mostly like a list, except that:

- it maintains a list of expected attributes (columns)
- it provides notification when items are added or removed

meta
Container for keys, source functions and remembered sorting.

Type *ListMeta*

sorted
whether list is currently sorted *by one of the list columns*. Sorting the list with a key function ("Python way") resets `sorted` to `False`.

Type bool

toolkit_ids
can be indexed in the same way as the nodelist, and gives the toolkit-specific identifier of the list/treeview node.

append(*value*)
S.append(value) – append value to the end of the sequence

children_source(*children_source*, *has_children_source=None*)
Sets the source for children of each list item, turning the list into a tree.

`children`, `has_children` follow the same semantics as other sources.

Resolving `children` should return an iterable that will be turned into an `ObsList` of its own.

`has_children` should return a truthy value that is used to decide whether to display the expander. If omitted, all nodes get the expander initially if children_source is set.

Children source only applies when the list of children is initially retrieved. Once the children are retrieved, source changes do not affect already-retrieved children anymore.

`has_children` is usually evaluated immediately, because the treeview needs to decide whether to display an expander icon.

clear() → None – remove all items from S

count(*value*) → integer – return number of occurrences of value

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

find(*item*)
Finds the sublist and index of the item.

Returns `(sublist: ObsList, idx:int)`.

If not found, raises ValueError.

Scans the whole tree for the item.

find_by_toolkit_id(*toolkit_id*)
finds the sublist and index of the item having the given toolkit id.

Returns `(sublist: ObsList, idx: int)`

If not found, raises ValueError.

Scans the whole tree for the item.

get_children(*idx*)
Get childlist of item at given idx, loading it if not already loaded.

has_children(*item*)

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*idx*, *item*)
S.insert(index, value) – insert value before index

item_mutated(*item*)
Call this when you mutated the item (which must be in this list) and want to update the GUI.

load_children(*idx*)
Retrieves the childlist of item at given idx.

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

retrieve(*item*, *column=''*)

reverse()
S.reverse() – reverse *IN PLACE*

selection
returns the sublist of all currently-selected items.

Raises RuntimeError if the nodelist is detached.

set_listener(*listener*)
Set listener that observes the list.

The listener can provide any or all of the following methods:

- `on_insert(idx, item, toolkit_parent_id) -> toolkit_id`: function to call for each inserted item
- **`on_replace(toolkit_id, item)`: function to call for replaced item** Replacement of item implies that children are "collapsed" again.
- `on_remove(toolkit_id)`: function to call for each removed item
- `on_load_children(toolkit_parent_id, sublist)`: function when children of a node are retrieved.
- **`on_get_selection()`: return the items selected in the GUI** Must return a List of (original) items.
- `on_sort()`: when list is reordered

`set_listener(None)` to reset listener.

sort(*key=None*, *ascending: bool = None*, *restore=False*)
Sort the list.

`key` can be a string, refering to the particular column of the listview. Use Empty String to refer to the anonymous column.

If `key` is a callable, the list is sorted in normal fashion.

Instead of specifying `key` and `ascending`, you can set `restore=True` to reuse the last applied sorting, if any.

sources(*_text=None*, ***kwargs*)
Alter the data binding for each column.

Takes the column names as kwargs and the data source as value; which can be:

- Empty string to retrieve str(obj)
- **String `"name"` to retrieve attribute `name` from the source object** (on attribute error, try to get as item)

- list of one item `['something']` to get item `'something'` (think of it as index without object)
- Callable `lambda obj: ..` to do a custom computation.

The `_text` argument, if given, is used to set the source for the "default" (anynomous-column) value.

class ascii_designer.list_model.**ListMeta**(*keys*)

holds the metadata of a ObsList.

copy()

get_observer(*name*)

retrieve(*obj*, *source*)

CHAPTER 3

Changelog

- v0.3.1: Qt Toolkit menu accelerators
- v0.3.0: menus added (TBD: accelerators for Qt menus); fix Qt List crash
- v0.2.0: rework of the list model

CHAPTER 4

Developers

The project is located at https://github.com/loehnertj/ascii_designer

This is a hobby project. If you need something quick, contact me or better, send a pull request.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Q

R

S

T