ASCII Designer Documentation

Release 0.3.1

Johannes Löhnert

Dec 04, 2022

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.

The `f_icon` property can be set to the name of an image file to use as window icon. Note that the supported formats depend on the toolkit and maybe also operating system. I recommend to use `.ico` on windows and `.png` on *nix.

Finally, there is the *`f_build()`* method, which does the actual form generation. It calls the *`f_on_build`* hook, that you might want to override to initialize controls.

1.3 Toolkit

Currently there are implementations for Qt and Tkinter toolkit. You need to decide which one to use. Before showing the first `AutoFrame`, use `set_toolkit(name)` to set the toolkit.

In particular, `set_toolkit` supports:

- `"qt"` for Qt toolkit (using qtpy).
- `"tk"` for Tkinter
- `"ttk"` also for Tkinter, but using ttk-themed widgets wherever possible.

If you use any native methods / properties, e.g. to set text box backgrounds, obviously changing the toolkit requires changing these parts of your code.

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.5 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

1.5.1 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.

1.5.2 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).

1.6 Value and event binding

1.6.1 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, which you can find it as `self["control_id"].variable` attribute on the control. For textual widgets (Entry, Combobox, etc) this will be a *GenericVar* instance, which you can consider a supercharged `StringVar`. See below for what features it brings.

1.6.2 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).

1.6.3 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). For your convenience, you can directly assign an AutoFrame subclass instance to the virtual value, which triggers building of the AutoFrame.

In any case, you can retrieve the new widget or the AutoFrame as new virtual value of the placeholder.

1.6.4 TK Textbox / Combobox conversion + validation

ASCII designer ships with its own supercharged Tk variable class, the *GenericVar*. It is used for Entry, Combobox, Label and Button widgets.

Without further setup, it behaves like a regular `StringVar`.

Type conversion is done using the *convert* property. `convert` is a callback that is used within `.get()` to convert the string value into something else. This also applies when accessing the value via the virtual attribute, or when the automatic handler is called.

For example, you can do the following:

```
class ConvertingFrame(AutoFrame):
    f_body = """
    |
     [entry_ ]
    """

    def f_on_build(self):
        self.f_controls["entry"].variable.convert = int
```

When retrieving the value via `self.entry`, you will get a bona-fide Integer.

If the converter fails, `GenericVar.get()` will return the special *Invalid* object.

For your convenience, there are also some "compositors" for common value restrictions. See *nullable*, *gt0* and *ge0*.

If needed, you can also setup a conversion in the opposite direction (Value to display text) using `variable.convert_set`. In most cases, the default (`str`) should suffice.

Conversion implies Validation of the input text: We consider the input valid if the converter does not raise an Exception. `GenericVar` allows you to couple a side effect to `get()` using the *validated_hook* property. It might have a slight smell to it, but is frequently very convenient. :-)

You can enable this on a per-class level by setting the `f_option_tk_autovalidate` attribute of your frame class to `True`. In this case, for all Entry and Combobox widgets, invalid-state will be updated whenever the converted value is retrieved. For non-ttk widgets, the foreground color will be changed to red if invalid. For ttk widgets, `invalid` state is set.

Note that this also applies to plain-string entryboxes - they will always be valid, meaning that `invalid` state will be reset on each access. If you want to do custom validation for a single control, you can "opt-out" by using: `self.f_controls["name"].variable.validated_hook = None`.

1.7 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. It is actually an instance of `ObsList`, which provides "events" for all changes to the list. Also you can make it into a tree by configuring `children_source` (see below).

There is a toolkit specific adapter between the `ObsList` object and the actual onscreen widget - the `ListBinding`. It interconnects list and widget events, and provides the mapping between list item (any object) and column values.

- With TK/TTk Toolkit, get the binding object by `self['widgetname'].variable`.
- With Qt Toolkit, get the binding object by `self['widgetname'].model()`.

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 can become detached if you replace the virtual value while keeping the old reference somehow. You can still use it like a normal python object, but it will not have an onscreen representation anymore. If you attached own event handlers, take care of detaching them.

The *`ListBinding.sources`* method of the binding is 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.

The simplemost way of using the List is this:

```
class SimpleList(AutoFrame):
    f_body = '''
        |
         [= Some Items]
    '''
    def f_on_build(self):
        # 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_on_build(self):
        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'],
        )
        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). E.g. if the user sorts the list view, the underlying `ObsList` changes order. In case of doubt, make a copy.

1.7.1 Editing

As of v0.4, a list column can be made editable by appending _ (underscore) to the column caption. Some default shortcuts (`F2`, `Return`, etc.) apply.

The `ListBinding.sources` setting for the column also determines how edits are processed.

- If set to a string value, the corresponding property is set.
- If set to a 1-item list, `setitem` is used. I.e. `object[<name>] = <value>`.
- If set to a callable, `fn(obj, val)` is called.

Especially in the latter case, you will want to split into getter and setter method. To achieve this, set the source to a 2-tuple of definitions. Example:

```
def my_setter(obj, val):
  obj.my_property = float(val)

self.my_list.binding.sources(my_column=('my_property', my_setter))
```

reads the value of `my_column` by taking `my_property`, but upon edit, converts the value to float.

If you use the Tk toolkit, instead of `ttk.Treeview` you will get a `tk_treeedit.TreeEdit` instance. This is a custom Tk widget providing the edit functionality as well as some more. Please refer to its documentation for details.

The `add`, `adchild` and `remove` actions, if permitted, are handled by the binding. `ListBindingTk` has a `factory` property which provides new items when `add` function is used.

In Qt toolkit, Add / Add Child / Remove functions are currently not provided as builtin function.

Note: Differences between Qt and Tk:

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.

In Tk, a custom editable list widget is provided. In Qt, the native editing capabilites are used.

1.7.2 Trees

Trees are created by using the *`ObsList.children_source`* method, which works similar to `ListBinding.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.

The list object now stores *two* things for each index: The item at this index, and the list of the item's children. This is a bit different to how trees are usually implemented. Particularly, there is no single "root" item, instead we have a root list.

To index within the tree, use index tuples:

- `tree[1]` gives you the second item of the root list.
- `tree[1, 2]` gives you the third item in the child list of the second item.
- `tree[1, 2, 44, 21]` goes 4 levels deep
- `tree[1, None]` gives you the *child list* of the second item. This is a separate *`ObsList`* instance.
- Again, `tree[1, 2, 44, None]` to retrieve a "deeper" level.

You can only access subitems that have been lazy-loaded already. Use `Obslist.get_children` to ensure loading.

Note: If you assign a non-`ObsList` value to a ListView virtual-value, it is converted into an `ObsList`. The `children_source` is taken over from the **previous** value. I.e. you can configure it once and then assign plain lists, retaining tree configuration. This is done for your convenience and for backward compatibility.

If on the other hand, you assign an `ObsList` instance as value, it is assumed that its `children_source` is already configured, and it won't be touched. This is because `children_source` is taken to be part of the data-model and not of the GUI binding.

1.7.3 Toolkit-native identifiers

If you handle toolkit-native events yourself, you will likely need to cope with "toolkit native" identifiers (TKinter item id or Qt `QModelIndex`, respectively). `ObsList` keeps track of the association between toolkit ID and actual list item for you.

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.

1.8 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...

1.9 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?

1.9.1 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.

1.9.2 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.

1.9.3 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.

1.9.4 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_on_build(self):
        # 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

        # # can be simplified to:
        # self.placeholder = Embedded()
        # # (later, get the instance from self.placeholder)

class Embedded(AutoFrame):
    f_body = '''
        |
         <another placeholder>
    '''
    def f_on_build(self):
        parent = self.another_placeholder.master
        self.another_placeholder = tk.Button(parent, text='3rd-party control')
```

1.9.5 Custom widget classes

The toolkit has a `widget_classes` property, listing the widget class to generate for each widget type (per the widget creation syntax). By changing this dictionary, you can make the toolkit return a custom subclass. For instance, you could use a `tk.Entry` subclass with custom functionality:

```
class MyEntry(tk.Entry):
    # Just add a demonstration property here
    special_property = True

class FrameWithCustomWidget(AutoFrame):
    f_body = """
      |
       [My text_ ]
    """

    def __init__(self):
        super().__init__()
        self.f_toolkit.widget_classes["textbox"] = MyEntry
        # f_build will now use the changed class (only in *this* frame!)

    def on_my_text(self, val):
        widget = self["my_text"]
```

(continues on next page)

(continued from previous page)

```
        print(widget.__class__.__name__) # "MyEntry"
        if widget.special_property:
            pass # do something special
```

These are the names of the widget classes:

- `label`
- `box`
- `box_labeled`
- `option`
- `checkbox`
- `slider`
- `multiline`
- `textbox`
- `treelist`
- `treelist_editable` (tk only)
- `combo`
- `dropdown`
- `button`
- `scrollbar` (tk only)

CHAPTER 2

Translating forms

ASCII designer has a custom translation system for forms. You can also add own content e.g. for dynamic texts.

Builtin translation covers:

- Widget and label texts (Key `<Class>.<widget_id>`)
- List column headers (Key `<Class>.<widget_id>.<column_id>`)
- Menu entries (Key `<Class>.<menuitem_id>`)
- Form title (Key `<Class>.f_title`)

Translations are stored as JSON files. The content is a simple key-value store.

Translation files can be stored either within some python package (recommended), or accessed via path.

Whichever folder you use, name the files:

- `default.json` for fallback translations (usually English)
- `<code>.json` e.g. `de.json`, `en.json`, ... for individual languages
- If required, add country-specific versions as `de_DE.json`, `de_CH.json`.

2.1 Loading translations from a package

Assume that the translations files are stored in `my_app` package, in a subfolder `locale`. I.e. the actual translation package is `my_app.locale`. Remember that in Python 3.3+, the `__init__.py` is not required anymore.

File structure:

```
my_app/
    __init__.py
    my_app.py
    ...etc...
    locale/
```

(continues on next page)

(continued from previous page)

```
        default.json
        de.json
        es.json
        ...etc...
```

To use translations, add the following code where you init your app:

```
from ascii_designer import AutoFrame, load_translations_json

# in Setup code
translations = load_translations_json("my_app.locale")
AutoFrame.f_translations = translations
```

2.1.1 Remarks

Language will be picked up from the OS. To force a language, use e.g. `load_translations_json("my_app.locale", language="es")`.

The `importlib.resource` loader will be used, meaning you can put package and translations in a zip file if needed. *`load_translations_json`* will not fail even if no translation file is found.

`translations` is a *`Translations`* instance. For the most part, it's just a dict matching the JSON's content.

Keys are formed as `<ClassName>.<widget_id>`. Additionally, List column headers are stored as `<ClassName>.<widget_id>.<column_id>`. It is recommended to use the recording feature to generate the file (see below).

By setting the class attribute `AutoFrame.f_translations`, Translations are shared among all `AutoFrame` subclasses. You can also set translations just for an individual form or even instance, by setting their respective `f_translations` class or instance attribute.

2.2 Loading translations from a file

Works nearly the same. Pass to *`load_translations_json`*:

- a `pathlib.Path`, or
- a string containing at least one slash `/` or backslash `\`.

Instead of the `importlib.resource` loader, normal file loading is used.

2.3 Generating or updating translation files

`Translations` has a special flag *`recording`* that you can set. The dictionary will then record all translation keys that are queried.

Extend your startup code as follows:

```
from ascii_designer import AutoFrame, load_translations_json, save_translations_json

# in Setup code
translations = load_translations_json("my_app.locale", language="")
AutoFrame.f_translations = translations
```

(continues on next page)

(continued from previous page)

```
translations.recording = True

# Show your app here
# After mainloop exits (last window closed):
save_translations_json(AutoFrame.f_translations, "default.json")
```

Note that the default language is forced when loading.

Run the app, and open all forms that need to be translated at least once. After exiting the app, the new translation file is saved.

Existing entries of the translation file are kept untouched. New entries are appended at the end.

Now, copy or move the file to your translations directory. Make a copy for each language to translate into, e.g `de.json`, ... Then edit the translations with any tool of choice.

2.4 To see if there are any translations missing in the GUI

Set *`Translations.mark_missing`* flag. All GUI strings missing in the translation file will be prepended with `$`.

2.5 Adding custom translations

Just add them to the JSON dictionary under whatever key(s) you like.

In a Form method, retrieve them by:

```
custom_text = self.f_translations.get("my_key", "Default text here")
```

Always use *`get`*. That way the key can be "captured" by the `recording` and `mark_missing` features.

CHAPTER 3

API documentation (autogenerated)

3.1 ascii_designer package

ASCII Designer: Library to generate Forms from ASCII Art... and then some.

AutoFrame is the main workhorse class. Subclass it and set the f_body and maybe further attributes.

Package overview:

- *autoframe* provides the *AutoFrame* class, which by itself contains the build logic and the automatic binding infrastructure.
- *toolkit* provides the widget generators and actual value and event binding. The base *ToolkitBase* contains the parsing function and lots of abstract methods.
- *toolkit_tk* and toolkit_qt provide actual implementations of the Toolkit interface.
- *ascii_slice* contains the text slicing algorithm used to cut text into grid cells.
- *list_model* contains the list-like value class for list view / tree view. The *ObsList* is toolkit-agnostic and has has lots of hooks where the GUI bindings (in the toolkit_ modules) connect to.

class ascii_designer.**AutoFrame**

Automatic frame.

class name is converted to title. Override with f_title.

Set window icon by giving an icon file's path in f_icon. Supported formats are OS-specific; recommended are .ico on Windows and .png on Unix.

Body definition with f_body, menu definition with f_menu.

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

To add initialization code, override *f_on_show*.

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_on_build()
Hook that is called after form has been built.

Override this to add custom initialization of widgets.

f_on_show()
Hook that is called when form is about to be shown on screen.

In contrast to f_on_build, this is called again if the form is closed and reopened.

f_option_tk_autovalidate = False
If True, tk/ttk Entry and Combobox are set up for automatic update of widget state when validated.

I.e. when value is retrieved or handler is called, the widgets foreground color (tk) or `invalid` state (ttk) is updated.

Opt-in, because it might interfere with user code if not expected.

f_show()
Bring the frame on the screen.

f_translations = {}
Translation dictionary.

This can be set per-form or globally on the AutoFrame class. We only actually need the `.get(key, default)` method.

Translation keys are formed as `<Class name>.<Widget ID>`.

Translations are used in `f_build` and `f_build_menu` functions. Currently there is no facility to re-translate after building the form.

quit()

ascii_designer.**set_toolkit**(*toolkit_name*, *toolkit_options=None*)
Set the toolkit to use and toolkit options.

Toolkit name can be `tk`, `ttk`, `qt`.

`toolkit_options` is a dictionary of toolkit specific global settings like font size or theme. See *ToolkitTk*, `ToolkitQt`.

class ascii_designer.**EventSource**
Event dispatcher class

You can register / unregister handlers via += and −= methods.

Handlers *may* return a result. If multiple handlers return results, last one is returned to the event's source.

exception ascii_designer.**CancelEvent**
Raise this in an event handler to inhibit all further processing.

class ascii_designer.**Invalid**
Sentinel object to signal that the read value was invalid.

`Invalid` will returned as itself (i.e. not as instance).

ascii_designer.**nullable**(*convert*)
Creates a converter that returns `None` on empty string, otherwise applies given converter.

```
>>> generic_var = GenericVar(tkroot, convert=nullable(float))
>>> generic_var.set("1.0")
>>> generic_var.get()
1.0
>>> generic_var.set("")
>>> generic_var.get()
None
>>> generic_var.set("foo")
>>> generic_var.get()
<class Invalid>
```

ascii_designer.**gt0**(*convert*)
Applies `convert`, then raises if value is not greater than 0.

`convert` must return something number-like.

```
>>> generic_var.convert = gt0(int)
```

ascii_designer.**ge0**(*convert*)
Applies `convert`, then raises if value is not greater or equal to 0.

`convert` must return something number-like.

```
>>> generic_var.convert = ge0(int)
```

class ascii_designer.**Translations**
Mostly off-the shelf python dict, except for two facilities to aid translation.

Translations should be retrieved via `.get(key, default)` method.

The class has the two additional properties *recording* and *mark_missing*.

- If `recording` is set to True, calls of *`get`* will add missing entries (i.e. *`get`* does the same as `setdefault`). By setting it and opening all forms once, you can collect all translation keys and default strings.
- If `mark_missing` is set and *`get`* finds a missing key, the given default value is prefixed with a `$` sign.

get (*key*, *default=None*)
Return the value for key if key is in the dictionary, else default.

mark_missing = False

recording = False

ascii_designer.**load_translations_json** (*package_or_dir='locale'*, *prefix=''*, *language=None*)
Locate and load translations from JSON file.

JSON file format is a simple key value store.

If given a package name, use the resource loading system. If given a dir, use file access.

The argument is interpreted as dir if:

- the string contains `/` or `\`
- the argument is a `pathlib.PurePath` instance.

Resource name is formed by the rule "<prefix>.<language>.json" (first dot is ommited if one of both is empty). If both prefix and language are empty, we look for `default.json`.

If the language is not given, the OS's UI language is used.

With the given or guessed language we look for an existing file:

- First we look for the exact language string (e.g. "de_DE.json")
- then we look for the first two letters of the language string ("de.json")
- then we look for emtpy language (i.e. default set).

If none of these exists, empty `Translations` object is returned.

ascii_designer.**save_translations_json** (*translations*, *path*)
Save translations to JSON file.

OVERWRITES existing file!

In contrast to `load_translations_json`, we only accept a path here.

3.2 ascii_designer.autoframe module

Warning: module 'ascii_designer.autoframe' undocumented

class ascii_designer.autoframe.**AutoFrame**
Automatic frame.

class name is converted to title. Override with `f_title`.

Set window icon by giving an icon file's path in `f_icon`. Supported formats are OS-specific; recommended are `.ico` on Windows and `.png` on Unix.

Body definition with `f_body`, menu definition with `f_menu`.

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

To add initialization code, override *f_on_show*.

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_on_build()
Hook that is called after form has been built.

Override this to add custom initialization of widgets.

f_on_show()
Hook that is called when form is about to be shown on screen.

In contrast to f_on_build, this is called again if the form is closed and reopened.

f_option_tk_autovalidate = False
If True, tk/ttk Entry and Combobox are set up for automatic update of widget state when validated.

I.e. when value is retrieved or handler is called, the widgets foreground color (tk) or `invalid` state (ttk) is updated.

Opt-in, because it might interfere with user code if not expected.

f_show()
Bring the frame on the screen.

f_translations = {}
Translation dictionary.

This can be set per-form or globally on the AutoFrame class. We only actually need the `.get(key, default)` method.

Translation keys are formed as `<Class name>.<Widget ID>`.

Translations are used in `f_build` and `f_build_menu` functions. Currently there is no facility to re-translate after building the form.

quit()

3.3 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

3.4 ascii_designer.toolkit module

> **Warning:** module 'ascii_designer.toolkit' undocumented

ascii_designer.toolkit.**set_toolkit**(*toolkit_name*, *toolkit_options=None*)
Set the toolkit to use and toolkit options.

Toolkit name can be `tk`, `ttk`, `qt`.

`toolkit_options` is a dictionary of toolkit specific global settings like font size or theme. See *ToolkitTk*, ToolkitQt.

ascii_designer.toolkit.**get_toolkit**()
Get toolkit instance as previously set.

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*, *translations=None*, *translation_prefix=''*)
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.

Supports automatic translation of widgets.

- translations is a dict-like object. We will use `.get(key, default)` to retrieve values.
- `translation_prefix` is a global prefix, so that you can use the same dict for multiple forms.
- For each widget that has `text`, we look up `<translation_prefix><id>` in the dict. If present, the content will be used instead of original `text`.
- For treelist, we will also look up `<translation_prefix><id>.<column_id>` and use it as column name, if found.

parse_menu(*parent*, *menudef*, *handlers*, *translations=None*, *translation_prefix=''*)
Parse menu definition list and attach to the handlers.

Translations work the same as for *parse*.

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

root(*title='Window'*, *icon=''*, *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*, *first_column_editable=False*)
treeview (also usable as plain list) Column is a list of column definitions: something with .id, .text, .editable attrs.

widget_classes = {'box': None, 'box_labeled': None, 'button': None, 'checkbox': No
Actual class to use for each widget type.

This allows you to override the actual widget with a custom subclass, if desired.

Obviously, this should be done *before* building the frame, e.g. in `__init__`.

The replacement must have the same init signature as the original widget class.

class ascii_designer.toolkit.**ListBinding**(*keys*, ***kwargs*)
Glue code to connect an ObsList to a GUI List widget.

Takes care of:

- Extracting column values from list items
- Remembering/applying GUI-sorting
- Mapping "model" events to GUI actions and vice-versa.

Properties:

keys (list of str) column keys

list (ObsList) the bound list

sort_key (str) column sorted by

sort_ascending (bool) sort order

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

factory (function() -> Any) Factory function for new items (on add). Signature might change in future releases. I am not sure right now what parameters might be useful.

Abstract base class. Override methods where noted.

list

on_get_selection()
ABSTRACT: query selected nodes from GUI

on_insert(*idx*, *item*, *toolkit_parent_id*)
ABSTRACT: Insert item in tree, return toolkit_id

on_load_children(*children*)
ABSTRACT: insert children into GUI tree

on_remove(*iid*)
ABSTRACT: remove item in GUI

on_replace(*iid*, *item*)
ABSTRACT: update GUI with changed item

on_sort(*sublist*, *info*)
reorder rows in GUI

Base implementation remembers sort_key, sort_ascending entries from info, and sets `sorted` flag if both are there.

Subclasses should call the base, and then update header formatting if appropriate.

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

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

`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.

store(*item*, *val*, *column=''*)

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

3.5 ascii_designer.toolkit_tk module

TK Toolkit implementation

class ascii_designer.toolkit_tk.**ToolkitTk**(**args*, *prefer_ttk: bool = False*, *setup_style=None*, *add_setup=None*, *font_size: int = 10*, *ttk_theme: str = ''*, *autovalidate: bool = False*, ***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.

Parameters

- **prefer_ttk** (*bool*) – Prefer ttk widgets before tk widgets. This means that you need to use the Style system to set things like background color etc.
- **setup_style** (*fn(root_widget) -> None*) – custom callback for setting up style of the Tk windows (font size, themes). If not set, some sensible defaults are applied; see the other options, and source of *setup_style()*.
- **add_setup** (*fn(root_widget) -> None*) – custom callback for additional setup. In contrast to `setup_style` this will not replace but extend the default setup function.
- **font_size** (*int*) – controls default font size of all widgets.
- **ttk_theme** (*str*) – explicit choice of theme. Per default, no theme is set if `prefer_ttk` is `False`; otherwise, `winnative` is used if available, otherwise `clam`.
- **autovalidate** (*bool*) – If True, generated widgets using `GenericVar` will set themselves up for automatic update of widget state when validated. This uses *GenericVar.validated_hook*. Affects Entry and Combobox. Can be changed afterwards.

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=''*)
Creates a TKinter frame or label frame.

A `.variable` property is added just like for the other controls.

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

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

setup_style (*root*)
Setup some default styling (dpi, font, ttk theme).

For details, refer to the source.

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*, *first_column_editable=False*)
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.

Columns can be marked editable by appending "_" to the name. If any column is editable, a *TreeEdit* is generated instead of the TreeView.

Returns the treeview widget (within the frame).

widget_classes_tk = {'box': <class 'tkinter.Frame'>, 'box_labeled': <class 'tkinter.

widget_classes_ttk = {'box': <class 'tkinter.ttk.Frame'>, 'box_labeled': <class 'tki

3.6 ascii_designer.toolkit_qt module

3.7 ascii_designer.i18n module

I18n support for ASCII Designer or other purposes.

Provides the working-data structure *Translations*, and functions to load and save translation files.

class ascii_designer.i18n.**Translations**
Mostly off-the shelf python dict, except for two facilities to aid translation.

Translations should be retrieved via `.get(key, default)` method.

The class has the two additional properties *recording* and *mark_missing*.

- If *recording* is set to True, calls of *get* will add missing entries (i.e. *get* does the same as `setdefault`). By setting it and opening all forms once, you can collect all translation keys and default strings.
- If *mark_missing* is set and *get* finds a missing key, the given default value is prefixed with a `$` sign.

get (*key*, *default=None*)
Return the value for key if key is in the dictionary, else default.

mark_missing = False

recording = False

ascii_designer.i18n.**load_translations_json** (*package_or_dir='locale'*, *prefix=''*, *language=None*)
Locate and load translations from JSON file.

JSON file format is a simple key value store.

If given a package name, use the resource loading system. If given a dir, use file access.

The argument is interpreted as dir if:

- the string contains / or \
- the argument is a pathlib.PurePath instance.

Resource name is formed by the rule "<prefix>.<language>.json" (first dot is ommited if one of both is empty). If both prefix and language are empty, we look for default.json.

If the language is not given, the OS's UI language is used.

With the given or guessed language we look for an existing file:

- First we look for the exact language string (e.g. "de_DE.json")
- then we look for the first two letters of the language string ("de.json")
- then we look for emtpy language (i.e. default set).

If none of these exists, empty Translations object is returned.

ascii_designer.i18n.**save_translations_json**(*translations*, *path*)

Save translations to JSON file.

OVERWRITES existing file!

In contrast to load_translations_json, we only accept a path here.

3.8 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 event callbacks for insert, replace, remove, sort.
- **a ListBinding:**
  - provides COLUMNS (key-value items) dynamically retrieved from each list item (using *retrieve* function from here)
  - remembers column and order to be used when sorting
  - has a notion of "selection" (forwarded to a callback)

class ascii_designer.list_model.**ObsList**(*iterable=None*, *binding=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
- it can be made into a tree by means of the children_source setting (see there).

If configured as tree, indexing happens via tuples:

- mylist[8] returns the 8th item at toplevel, as usual
- mylist[3, 2] returns the 2nd item of the 3rd items' children.

- `mylist[3, 2, 12, 0]` goes 4 levels deep
- `mylist[3, None]` can be used to retreive the list of children of item 3, instead of a specific child item.

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

Events:

- `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(sublist, info)`: when list is reordered

For `on_insert`, the handler should return the toolkit_id. This is an unspecified, toolkit-native object identifier. It is used in the other events and for `find_by_toolkit_id`. Its purpose is to allow easier integration with toolkit-native events and methods.

`on_sort`: Info argument is a dict containing custom info, e.g. column that was sorted by.

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

`binding`
Weak reference to the binding using this list.

If list is attached to multiple Bindings, last one wins. (TODO: make it a list)

`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.

find2(*item*)
Finds the tuple-index of the item.

Returns `idx: Tuple[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.

find_by_toolkit_id2(*item*)
Finds the tuple-index of the item having the given toolkit id.

Returns `idx: Tuple[int]`.

If not found, raises ValueError. Scans the whole tree for the item.

get_children(*idx_tuple*)
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_tuple*, *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_tuple*)
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.

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

selection
returns the sublist of all currently-selected items.

Returns empty list if no handler is attached.

sort(*key=None*, *reverse=False*, *info=None*, *restore=False*)
Sort the list.

`info` is optional information to be passed on to on_sort.

Set `restore` to reuse key and info from last `sort` call.

sources(*_text=None*, ***kwargs*)
Forwards to `ListBinding.sources` - see there.

ascii_designer.list_model.**retrieve**(*obj*, *source*)
Automagic retrieval of object properties.

If `source` is empty string, return `str(obj)`.

If `source` is a plain string (identifier), use `getattr` on `obj`; on error, try `getitem`.

If `source` is a list with a single string-valued item, use `getitem` on `obj`.

If `source` is a callable, return `source(obj)`.

If `source` is a 2-tuple, use the first item ("getter") as above.

ascii_designer.list_model.**store**(*obj*, *val*, *source*)
Automagic storing of object properties.

If ''source'' is a plain string (identifier), use `setattr` on `obj`.

If `source` is a list with a single string-valued item, use `setitem` on `obj`.

If `source` is a callable, call it as `fn(obj, val)`.

If `source` is a 2-tuple of things, use the second item ("setter") as above.

Note: *store* is not fully symmetric to its counterpart *retrieve*.

- Empty source can not be used for *store*
- Plain string source will always use `setitem` without fallback.

If you use a single callable as source, it must be able to discern between "getter" and "setter" calls, e.g. by having a special default value for the second parameter.

3.9 ascii_designer.tk_treeedit module

`ttk.TreeView` control augmented by editing capabilities.

For basic information, see official Tkinter (`ttk`) docs.

The following additional functionality is provided:

- Mark column as editable using *TreeEdit.editable*.
- allow= parameter to specify legal structural operations.

`allow` is a list of strings or a comma-separated string. It can contain any of:

- `add` to allow adding new items (anywhere)
- `addchild` to allow insertion of child items
- `remove` to allow deletion of items.

For each allowance, the corresponding control is shown, and the keybinding is activated.

The following bindings / behaviors are built-in. Generally, value is submitted on close, except if Escape key is used.

Treeview:

- Dblclick: open edit on col
- Scroll: Take focus (close edit)
- Resize: close edit box

- F2: open first edit of row
- Ctrl+plus, Insert: add item (if enabled, see below)
- Ctrl+asterisk: add child (if enabled)
- Ctrl+minus,
- Delete: remove item (if enabled)

Edit box:

- Lose Focus: close
- Return: close
- Escape: close without submitting
- Shift+enter,
- Down arrow: Close + edit same column in next row
- Tab,
- Shift+Right arrow: close + edit next column (or 1st col in next row)
- Shift+Tab,
- Shift+Left arrow: like Tab but backwards
- Up arrow: Close + edit same col in prev row

Events:

These are properties of the TreeeEdit control. Use `treeedit.<property> += handler``to bind a handler, ``-=` to unbind it.

- `on_cell_edit(iid, columnname, cur_value)` when editor is opened
- `on_cell_modified(iid, columname, new_value)` when editor is closed
- `on_add(iid)`: before item is inserted after (iid).
- `on_add_child(iid)`: before child is inserted under (iid).
- `on_remove(iid)`: before child is deleted

Note: `on_cell_modified`, `on_add`, `on_add_child`, `on_remove` are fired immediately before the respective action takes place in the widget.

Your handler can return `False` to indicate that the widget content shall not be modified; e.g. if the action is forbidden or you took care of updating the tree yourself. Note that `None` is counted as `True` result in this case.

TODO: # reorder # custom editor types (button, checkbox, combo, ... pass own widget(s)) # copy/paste # have a handler for after-item-insertion with the actual iid as param (we don't need it currently)

class ascii_designer.tk_treeedit.**TreeEdit**(*master*, *allow=None*, **args*, ***kwargs*)

see module docs

advance(*direction='right'*)

switch to next cell.

`direction` can be left, right, up, down.

If going left/right beyond the first or last cell, edit box moves to the previous/next row.

advance_down(*ev=None*)

advance_left(*ev=None*)

advance_right(*ev=None*)

advance_up(*ev=None*)

allow
Allowed structural edits (add, delete, addchild).

Pass the allowed actions as list of strings or comma-separated string.

Can be updated during operation.

autoedit_added
Automatically begin editing added items yes/no

begin_edit(*iid*, *column*)
Show edit widget for the specified cell.

begin_edit_row(*ev*)
Start editing the first editable column of the focused row.

close_edit(*ev=None*, *cancel=False*)
Close the currently open editor, if any.

del_item(*ev=None*)
Trigger deletion of focused item.

editable(*column*, *editable=None*)
Query or specify whether the column is editable.

Only accepts Column Name or `'#0'`.

editbox_bindings = [('<FocusOut>', 'close_edit'), ('<Return>', '_close_edit_refocus'),

ins_child_item(*ev=None*)
Trigger insertion of new child item

ins_item(*ev=None*, *child=False*)
Trigger insertion of a new item.

list_bindings = [('<Double-Button-1>', '_dblclick'), ('<F2>', 'begin_edit_row'), ('<4>

3.10 ascii_designer.tk_generic_var module

Tkinter generic variable.

Comes with tools:

- `Invalid` special value
- `gt0`, `ge0`, `nullable` converter compositors

Also, it sports the `validated_hook` property for all-in-one validation and conversion.

class ascii_designer.tk_generic_var.**GenericVar**(*master=None*, *value=None*, *name=None*, *convert=<class 'str'>*, *convert_set=<class 'str'>*, *validated_hook=None*)
Generic variable.

A conversion function can be specified to get a "native" value from the string representation and vice-versa. By default, behaves like a StringVar.

master can be given as master widget. value is an optional value (defaults to "") name is an optional Tcl name (defaults to PY_VARnum). If name matches an existing variable and value is omitted then the existing value is retained.

convert = None
Gives the conversion to apply when get-ting the value.

convert takes the string (widget content) as single argument and returns a converted value. Any Exceptions will be caught (see *get*).

convert_set = None
Gives the function to apply to a value passed in to set.

In most cases the default (str) will be sufficient.

get()
Return converted value of variable.

If conversion failed, returns *Invalid*.

set(*value*)
Set the variable to VALUE.

validated_hook
If set, adds a side-effect to get depending on whether the value was valid or not.

It shall be a callback taking a single bool argument valid; which is True or False depending on whether convert raised an exception.

If you set validated_hook to a Tk widget, we will automatically convert it into a callback:

- For a Tk widget, foreground color will be set to red / original color depending on valid.
- For a Ttk widget, invalid state will be set/reset. Note that in most themes, by default no visual change happens, unless you configured an appropriate style map.

class ascii_designer.tk_generic_var.**Invalid**
Sentinel object to signal that the read value was invalid.

Invalid will returned as itself (i.e. not as instance).

ascii_designer.tk_generic_var.**ge0**(*convert*)
Applies convert, then raises if value is not greater or equal to 0.

convert must return something number-like.

```
>>> generic_var.convert = ge0(int)
```

ascii_designer.tk_generic_var.**gt0**(*convert*)
Applies convert, then raises if value is not greater than 0.

convert must return something number-like.

```
>>> generic_var.convert = gt0(int)
```

ascii_designer.tk_generic_var.**nullable**(*convert*)
Creates a converter that returns None on empty string, otherwise applies given converter.

```
>>> generic_var = GenericVar(tkroot, convert=nullable(float))
>>> generic_var.set("1.0")
>>> generic_var.get()
1.0
>>> generic_var.set("")
```

(continues on next page)

(continued from previous page)

```
>>> generic_var.get()
None
>>> generic_var.set("foo")
>>> generic_var.get()
<class Invalid>
```

3.11 ascii_designer.event module

Provides an event dispatcher class.

exception ascii_designer.event.**CancelEvent**
Raise this in an event handler to inhibit all further processing.

class ascii_designer.event.**EventSource**
Event dispatcher class

You can register / unregister handlers via += and −= methods.

Handlers *may* return a result. If multiple handlers return results, last one is returned to the event's source.

CHAPTER 4

Changelog

- v0.5.2:
  - Add translation support.
- v0.5.1:
  - TK: add `add_setup` toolkit option (similar to `setup_style` but keeps the defaults)
  - `ObsList`: add tuple indexing (experimental).
- v0.5.0:
  - TK/TTK: Add `GenericVar` supporting automatic conversion and validated-hook; opt-in automatic update of control's invalid state. See *TK Textbox / Combobox conversion + validation*
  - introduce `Toolkit.widget_classes` attribute, allowing injection of custom widget subclasses
  - export `EventSource`, `CancelEvent`, `Invalid`, `nullable`, `gt0`, `ge0` on package-level
- v0.4.4:
  - Fix edit-widget placement issues in Tk Treeedit
  - Treeedit subwidgets are now ttk (i.e. stylable)
  - Treeedit `autoedit_added` property
  - Fix icon "inheritance" on Windows
- v0.4.3:
  - Fix `f_icon` not working on Windows (with `.ico` files).
  - Tk (choice only) dropdown: fire event immediately on selection, not on `FocusOut` / `Return` press
  - Change virtual-value handling of placeholders. Placeholder "remembers" and returns what was assigned to it. Also, when assigned an unbuilt AutoFrame instance, it will automatically build it, reducing boilerplate for the typical embedding case.
- v0.4.2:
  - Fix 2 issues in ListBindingTk related to editable lists.

- v0.4.1:
  - Make tk_treeedit key bindings a class property (list) that can be modified. Fix crash on Windows due to nonexisting Keysym.
- v0.4.0:
  - Refactor internal implementation of List Binding. Architecture simplified to *ObsList* (data model) and *ListBinding* classes.
  - Add basic editing capabilites to List / Tree view.
- v0.3.4:
  - Add `setup_style` parameter to *ToolkitTk* for custom style setup
  - Fix #6: for some Tk controls, value change via code would trigger the control's autoconnected handler.
- v0.3.3:
  - Qt now depends on qtpy, not PyQt4 anymore.
  - `set_toolkit` can set Options. Add `font_size` and `ttk_theme` option for Tkinter.
  - Can set window icon by f_icon property
- v0.3.2:
  - Add "ttk" toolkit (which is actually *ToolkitTk* with a new option). YMMV.
  - Add *f_on_build* and *f_on_show* hooks
  - Fix several bugs in control parser, esp. concerning labels.
- 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 5

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 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

N

O

P

Q

R

S

T

V

W