QtBinder Documentation

Release 0.1.2

Enthought, Inc.

February 16, 2016

Contents

QtBinder thinly wraps Qt widgets with Traits.

The main goal of QtBinder is to provide a way to build Qt UIs with Traits models with an emphasis on transparency and flexibility. The core is the `Binder` class that automatically wraps a `QObject` and exposes its properties, signals, and slots as traits. Subclasses of a particular `Binder` can add traits and methods to customize the widget and expose useful patterns of communication between UI and model over the raw Qt API.

`Binder` widgets can be used inside a Traits UI `View` using a special `Item` called `Bound`. `Binder` widgets can be bound to model traits using *binding expressions*.

CHAPTER 1

When do I use QtBinder over Traits UI?

The two major pain points of Traits UI are getting widgets laid out precisely the way you need them to and customizing the behavior of editors in ways not intended by the original author. QtBinder addresses the layout problem by providing access to all of the layout tools that raw Qt has. It is even possible to lay out widgets in Qt Designer and attach the appropriate *`Binder`* to each widget.

`Bound` can be used to replace one normal `Item` in a Traits UI `View`, or by using a hierarchical layout *`Binder`*, it can replace some or all of what you would otherwise use normal Traits UI `Groups` for layout. You can use as much or as little of QtBinder as you need. It is easy to spot-fix the behavior of just one editor by replacing it with a `Binder` and leave the rest of the `View` alone. You do not have to wait until QtBinder has replicated the functionality of all Traits UI editors before using it to solve smaller problems.

Contents

2.1 Core Principles

1. **Value-added wrapping**: Custom *Binder* classes should only manually wrap the Qt API when it adds value. For example, translating Qt enums one-to-one to an ad hoc toolkit-neutral form does not add value. *Binder* can automatically wrap all Qt properties, signals, and slots. This means that a user of the custom subclass can access everything that the Qt widget exposes even if the author did not think to expose it. Value-added wrapping encapsulates patterns of communication and coordinates multiple moving pieces internal to the widget to expose a bindable Traits API.
2. **Thin, transparent wrapping**: This is a library for *using* Qt to build UIs, not hide it behind a toolkit-neutral abstraction.
3. **Small core**: The core should remain tiny so that it can be understood and traced through by users of QtBinder who are debugging their code.
4. **Graded transition from Traits UI**: *Bound* is a straightforward Traits UI `Item` that can be used wherever any other `Item` could be used in Traits UI. It can be used in a very focused manner to fix one or two places where the extra flexibility of QtBinder is necessary and ignored elsewhere. It can also be used to provide the whole `View` when desired. Use of QtBinder should not be held up because we have not added enough value-added widgets yet.
5. **Bind to existing instances**: All *Binder* classes can either instantiate their underlying `QWidget` or be provided an existing one. This allows us to lay out an entire UI in Qt Designer, instantiate it from the `.ui` file, then attach the desired *Binder* to individual widgets inside of it.
6. **Do one thing well**: Custom *Binder* subclasses should attempt to encapsulate one particular pattern of using their wrapped widget. It should not try to switch between different patterns based on configuration (unless if the intended pattern requires that the widget switch behaviors live). The logic needed to synchronize the widget state with the model state can sometimes get hairy. Dealing with multiple patterns conditionally complicates this part of the code, which makes it harder to customize for new purposes.
7. **Pay for what you use**: *Binder* wraps all Qt signals, but it will only connect to them and incur the cost of converting the signal values to Python objects when a Traits listener is attached to the signal trait.

2.2 Traits UI Integration

The *Bound* class is a Traits UI `Item` that can be used to place a *Binder* widget into a Traits UI `View` and bind it to traits on the model or `Handler`. It comes with its own `Editor` that knows how to set up the *Binder* and use it as the control for the `Item`.

The `Bound` constructor takes a `Binder` instance and some bindings. Bindings are either instances of `Binding` subclasses or, more conveniently, specially-formatted strings that will be parsed to `Binding` subclass instances.

```
traits_view = View(
    Bound(
        HBoxLayout(
            Label(id='label'),
            LineEdit(id='edit', placeholderText=u'An integer'),
        ),
        Factory('edit.validator', QtGui.QIntValidator),
        'label.text << handler.label',
        'edit.text := object.text',
        'spacing = 5',
    ),
)
```

This example `View` succinctly demonstrates most of the Traits UI features. The `HBoxLayout` is a `Binder` that transparently wraps the `QHBoxLayout` Qt layout object. It is slightly customized with a constructor that lets you declare the child widgets by passing `Binder` objects. Thus you can build most typical layouts using a hierarchy of layout and widget `Binder` objects. `Binder` constructors can take an `id` keyword argument that sets a name for the `Binder` that should be unique to the tree of `Binder` objects it is in. This name will be used to refer to that `Binder` in the bindings that follow. Other traits that proxy Qt properties can also be set in the `Binder` constructor. They will be assigned when the underlying `QObject` is assigned to the `Binder`.

Following the root `Binder` is a list of `Binding` strings or objects. These follow a pattern of `'binder_trait <operator> model_trait_or_expression'`. On the left of the operator is either the name of a trait on the root `Binder` (e.g. `spacing` refers to the `HBoxLayout.spacing` property) or a dotted reference to a trait on a descendant `Binder` that has provided an explicit `id` (e.g. `edit.text` refers to the `LineEdit.text` property).

On the right side of the operator is an expression evaluated in the Traits UI context. For a `Binding` that writes back to the model (`:=`/`SyncedWith` and `>>`/`PushedTo`), this is restricted to a simple extended trait reference; i.e. `object.foo.bar` but not `object.foo.bar + 10`. This context starts with the Traits UI context (i.e. has `object` and `handler` at a minimum) and is extended with any `Binder` in the tree with a non-empty `id`. For `<<`/`PulledFrom`, the expression will be parsed for extended trait references and the binding will be evaluated whenever it changes. For example, `format(handler.template, object.child.value)` will re-evaluate and assign to the left-hand side whenever `handler.template` OR `object.child.value` changes.

Note: Annoyingly, at the moment we cannot detect when such a dotted reference has a non-terminal non-`HasTraits` object. In the example above, `handler.template.format(object.child.value)` would cause an error because `handler.template` is a string, not a `HasTraits` object to which a listener can be attached.

There are four operators that can be used in the string representations of `Binding` objects:

- `=` or `SetOnceTo`: Set a value once. This evaluates the right-hand side once when the binding is established. No notifications will be sent afterwards.
- `<<` or `PulledFrom`: Pull values from the model. This evaluates the right-hand side once when the binding is established and whenever any traits used in the expression fire a change notification.
- `>>` or `PushedTo`: Push values from the `Binder` to the model. When the `Binder` trait on the left-hand side changes, this will assign the new value to the attribute referenced on the right-hand side. No value is assigned on initialization.
- `:=` or `SyncedWith`: A combination of `PulledFrom` and `PushedTo` to synchronize a binder trait with a model trait. Because the right-hand side of `PushedTo` is restricted to plain attribute references, so is this. Like `PulledFrom`, the right-hand side will be evaluated when the binding is established and assigned to the

left-hand side to initialize it.

And the last *Binding* cannot be put into string form:

- *Factory*: Call the provided function once when the binding is established, and set the value. No notifications will be sent afterwards.

Bindings which initialize a value (i.e. *SetOnceTo*/=, *PulledFrom*/<<, *SyncedWith*/:=, and *Factory*) will be evaluated in the order in which they are specified. This can be important for initializing some Qt objects. For example, setting up validator properties before assigning the value.

Bound takes the following optional keyword arguments:

label [unicode] Like the normal Item label argument, except that if one is not provided, then *Bound* will set show_label=False. Since the *Bound* Item is not exclusively associated with any single trait like other Traits UI Items are, the default Traits UI behavior of using the trait name as a label is not useful.

extra_context [dict] Any extra objects that should be added to the context used to evaluate the right-hand-side of bindings.

configure [function with signature configure(binder, context)] A function to call after the root *Binder* has been constructed and the bindings established but before display. It will be passed the root *Binder* and the context dictionary. This can be used to do customizations using the raw Qt API that may not be achievable using bindings alone.

stylesheet [unicode] A Qt stylesheet applied to the root control.

button_groups [dict naming *ButtonGroup* objects] Collect buttons in the UI into named, bindable groups that will be added to the context.

2.3 To Do

2.3.1 Short Term

- Demonstrate some fancier use cases that Traits UI does not handle well, like double-ended sliders made in Chaco (with histogram of a dataset being shown underneath).
- Bikeshed all the names.

2.3.2 Long Term

- Develop a reasonable story for the reverse wrapping: wrapping Traits object in the Qt item models API. Traits UI's TabularAdapter is a reasonable start, but it misses a lot of opportunities to be ideal according to our *Core Principles*.
- Have sufficient replacements for all common Traits UI editors and the ways that we have hacked them. The following are those that are sufficiently complicated that a configured raw widget *Binder* would not suffice (or are not otherwise covered elsewhere here).
  - TextEditor: we still need a LineEdit customization that converts specific Python objects (floats, ints, whatevers) to/from strings and validates the same.
  - EnumEditor: there are two distinct use cases, to select from a list of specific items or to allow write-in values with some recommended choices. Keep those use cases separate.

  - `BoundsEditor`: don't reuse the implementation. Use `(low, high)` tuples for both the value and the outer range. It's easier to handle the events that way. Also, we want to be able to grab the middle of the slider to move the whole range and not just each end independently. Keep it interface-compatible with the Chaco double-ended slider.
  - `ColorEditor`: design a nicer UI than the current one.
  - `DateEditor`
  - `TimeEditor`
  - `DirectoryEditor`
  - `FileEditor`
  - `SetEditor`

  As you can see, it's not that much.

- Inspect a *`Binder`* hierarchy and write it out as a Qt Designer `.ui` file so you can prototype the *`Binder`* using the simple declarative syntax, then tweak it quickly to look excellent for production.
- Wrap QtQuick components. QML is going to be particularly good for heavily customized table widgets.

2.3.3 Un-goals

- Other toolkits.
- Constraint-based layout. It can be useful for some advanced use cases, but is largely unnecessary for almost all of our use cases. It can be hard to debug without the right tooling (a la Apple), and the simple use cases sometimes fail inscrutably. Of course, it can be added independently as a `QLayout` if needed.

2.4 API Reference

2.4.1 `qt_binder.binder`

class `qt_binder.binder.`**`Binder`**(**args*, ***traits*)

Bases: `traits.has_traits.HasStrictTraits`

Traited proxy for a `QObject` class.

The default proxy traits will be automatically assigned by inspecting the Qt class specified in the *`qclass`* class attribute. Since this inspection process can be time consuming, compared to normal class construction, this will only be done the first time the `Binder` class is instantiated.

For those traits that proxy a Qt `Signal` (or property that has a `Signal`), the Qt signal connection will only be made once a **Traits** listener is attached to the proxy trait.

The *`qobj`* can only be assigned once in the `Binder's` lifetime.

`qclass`

The `QObject` **class** that is going to be wrapped by this class.

`qobj` **= Instance(QtCore.QObject)**

The Qt object instance that is wrapped by the *`Binder`* instance.

`loopback_guard` **= Instance(LoopbackGuard, args=())**

The loopback guard.

id = Str()
An ID string, if any. It should be a valid Python identifier.

construct (**args*, ***kwds*)
Default constructor that will automatically instantiate `qclass`.

configure ()
Do any configuration of the `qobj` that is needed.

dispose ()
Remove any connections and otherwise clean up for disposal.

This does not mark any Qt objects for deletion.

---

class qt_binder.binder.**Composite** (**args*, ***traits*)
Bases: *qt_binder.binder.Binder*

Base class for Binders that hold other Binders as children.

Their `QObjects` may or may not have a similar parent-child relationship. The `Composite` is responsible for constructing its children, configuring them, and disposing of them.

child_binders = Property(List(Instance(Binder)))
The child `Binder` instances. This will typically be a Property returning a list of `Binders` that are attributes.

configure ()
Do any configuration of the `qobj` that is needed.

dispose ()
Remove any connections and otherwise clean up for disposal.

This does not mark any Qt objects for deletion.

---

class qt_binder.binder.**NChildren** (**args*, ***traits*)
Bases: *qt_binder.binder.Composite*

Base class for Composite Binders that have arbitrary unnamed children.

child_binders = List(Instance(Binder))
Any children. It will be filtered for Binders.

---

class qt_binder.binder.**QtTrait** (**args*, ***metadata*)
Bases: `traits.trait_handlers.TraitType`

Base class for Qt proxy traits on *Binder* classes.

Each subclass should override *get()* and *set()*. All *QtTrait* subclasses are property-like traits.

If there is a Qt `Signal` that should be connected to to propagate notifications, assign it to the `signal` attribute. The Qt `Signal` will only be connected to when a Traits listener is attached to this trait.

get (*object*, *name*)
Get the value of this trait.

set (*object*, *name*, *value*)
Set the value of this trait and notify listeners.

connect_signal (*object*, *name*)
Connect to the Qt signal, if any.

disconnect_signal (*object*, *name*)

Disconnect from the Qt signal, if any.

---

class `qt_binder.binder.`**`QtProperty`** (*meta_prop*, ***metadata*)

Bases: *`qt_binder.binder.QtTrait`*

Proxy trait for a Qt static property.

Pass in a `QMetaProperty` from the `QMetaObject`.

get (*object*, *name*)

Get the value of this trait.

set (*object*, *name*, *value*)

Set the value of this trait and notify listeners.

If there is a Qt `Signal` for this property, it will notify the listeners. If there is not one for this property, this method will explicitly send a notification.

---

class `qt_binder.binder.`**`QtDynamicProperty`** (*default_value=None*, ***metadata*)

Bases: *`qt_binder.binder.QtTrait`*

A Qt dynamic property added to the `QObject`.

The dynamic property will be created on the `QObject` when it is added to the *`Binder`*. The default value given to this trait will be the initial value. It should be an object that can be passed to `QVariant`.

Because most dynamic properties will be added this way to support Qt stylesheets, by default when the property is assigned a new value, the `QObject` associated with the `Binder` (which should be a `QWidget`) will be made to redraw itself in order to reevaluate the stylesheet rules with the new value. Turn this off by passing `styled=False` to the constructor.

get (*object*, *name*)

Get the value of this trait.

set (*object*, *name*, *value*)

Set the value of this trait and notify listeners.

---

class `qt_binder.binder.`**`QtGetterSetter`** (*getter_name*, *setter_name=None*, ***metadata*)

Bases: *`qt_binder.binder.QtTrait`*

Proxy for a getter/setter pair of methods.

This is used for `value()/setValue()` pairs of methods that are frequently found in Qt, but which are not bona fide Qt properties.

If the names follow this convention, you only need to pass the name of the getter method. Otherwise, pass both.

get (*object*, *name*)

Get the value of this trait.

set (*object*, *name*, *value*)

Set the value of this trait and notify listeners.

---

class `qt_binder.binder.`**`QtSlot`**(*meta_method*, ***metadata*)

Bases: *`qt_binder.binder.QtTrait`*

Proxy for a Qt slot method.

In general use, this trait will only be assigned to. If the slot takes no arguments, the value assigned is ignored. If the slot takes one argument, the value assigned is passed to the slot. If the slot takes more than one argument, the value assigned should be a tuple of the right size.

As a convenience, getting the value of this trait will return the slot method object itself to allow you to connect to it using the normal Qt mechanism.

The constructor should be passed the `QMetaMethod` for this slot.

`get`(*object*, *name*)

Get the underlying method object.

`set`(*object*, *name*, *value*)

Set the value of this trait.

See *`QtSlot`* for details on how the value is processed.

---

class `qt_binder.binder.`**`QtSignal`**(*meta_method*, ***metadata*)

Bases: *`qt_binder.binder.QtSlot`*

Proxy for a Qt signal method.

In general use, this trait will only be listened to for events that are emitted internally from Qt. However, it can be assigned values, with the same argument semantics as *`QtSlot`*. Like *`QtSlot`*, getting the value of this trait will return the signal method object itself for you to connect to it using the normal Qt mechanism.

The constructor should be passed the `QMetaMethod` for this signal.

`set`(*object*, *name*, *value*)

Emit the signal with the given value.

See *`QtSlot`* for details on how the value is processed.

---

class `qt_binder.binder.`**`Default`**(*value*)

Bases: `object`

Specify a default value for an automatic QtTrait.

---

class `qt_binder.binder.`**`Rename`**(*qt_name*, *default=<undefined>*)

Bases: `object`

Specify that an automatic QtTrait be renamed.

Use at the class level of a *`Binder`* to rename the trait to something else.

For *`QtSlot`* traits with multiple signatures, only the primary part of the name (without the mangled type signature) needs to be given.

Since one cannot use both a *`Default`* and *`Rename`* at the same time, one can also specify the default value here.

2.4.2 `qt_binder.binding`

class `qt_binder.binding.`**`Binding`**(*left*, *right*)

Bases: `object`

Interface for a single binding pair.

classmethod **`parse`**(*obj*)

Parse a binding expression into the right *`Binding`* subclass.

`bind`(*binder*, *context*)

Perform the binding and store the information needed to undo it.

`unbind`()

Undo the binding.

---

class `qt_binder.binding.`**`SetOnceTo`**(*left*, *right*)

Bases: *`qt_binder.binding.Binding`*

Evaluate values once.

The right item of the pair is a string that will be evaluated in the Traits UI context once on initialization.

Mnemonic: `binder_trait is set once to expression`

---

class `qt_binder.binding.`**`Factory`**(*left*, *right*)

Bases: *`qt_binder.binding.Binding`*

Call the factory to initialize a value.

The right item of the pair is a callable that will be called once on initialization to provide a value for the destination trait.

---

class `qt_binder.binding.`**`PulledFrom`**(*left*, *right*)

Bases: *`qt_binder.binding.Binding`*

Listen to traits in the context.

The right item of each pair is a string representing the extended trait to listen to. The first part of this string should be a key into the Traits UI context; e.g. to listen to the `foo` trait on the model object, use `'object.foo'`. When the `foo` trait on the model object fires a trait change notification, the `Binder` trait will be assigned. The reverse is not true: see *`PushedTo`* and *`SyncedWith`* for that functionality.

Mnemonic: `binder_trait is pulled from context_trait`

---

class `qt_binder.binding.`**`PushedTo`**(*left*, *right*)

Bases: *`qt_binder.binding.Binding`*

Send trait updates from the `Binder` to the model.

The right item of each pair is a string representing the extended trait to assign the value to. The first part of this string should be a key into the Traits UI context; e.g. to send to the `foo` trait on the model object, use `'object.foo'`. When a change notification for `binder_trait` is fired, `object.foo` will be assigned the sent object. The reverse is not true: see *`PulledFrom`* and *`SyncedWith`* for that functionality.

Mnemonic: `binder_trait is sent to context_trait`

class `qt_binder.binding.`**`SyncedWith`**(*left*, *right*)

Bases: *`qt_binder.binding.PulledFrom`*, *`qt_binder.binding.PushedTo`*

Bidirectionally synchronize a `Binder` trait and a model trait.

The right item of each pair is a string representing the extended trait to synchronize the binder trait with. The first part of this string should be a key into the Traits UI context; e.g. to synchronize with the `foo` trait on the model object, use `'object.foo'`. When a change notification for either trait is sent, the value will be assigned to the other. See *`PulledFrom`* and *`PushedTo`* for unidirectional synchronization.

Mnemonic: `binder_trait is synced with context_trait`

2.4.3 `qt_binder.bound_editor`

class `qt_binder.bound_editor.`**`Bound`**(*binder*, **bindings*, ***kwds*)

Bases: `traitsui.item.Item`

Convenience `Item` for placing a `Binder` in a `View`.

class `qt_binder.bound_editor.`**`TraitsUI`**(*item=None*, ***traits*)

Bases: *`qt_binder.binder.Binder`*

Place a Traits UI `Item` into a `Bound` layout.

The automatically-added traits are only those for `QWidget`, not whatever widget the root control of the `Item` may turn out to be. This *`Binder`* can only be used in the context of a *`Bound`* layout because it needs to be specially recognized and initialized.

`item` **= Instance(Item)**

The Traits UI Item to display. Any label is ignored.

`initialize_item`(*ui*)

Initialize the item using the Traits UI `UI` object.

2.4.4 `qt_binder.raw_widgets`

Mostly automated wrappers around all of the `QWidgets` and `QLayouts` provided in `PySide.QtGui`. Generally, the *`Binder`* is named by dropping the leading `Q`. Only a few of these are minimally customized when it is necessary to make them useful. Only those are documented here. The Qt API reference should be consulted for details of what properties, signals, and slots are defined.

`qt_binder.raw_widgets.`**`binder_registry`**

The global *`TypeRegistry`* mapping PySide/PyQt types to their default *`Binder`* class.

class `qt_binder.raw_widgets.`**`ComboBox`**(**args*, ***traits*)

Bases: *`qt_binder.binder.Composite`*

Customized to exposed the line-edit widget as a child *`Binder`*.

`qclass`

lineEdit_class
alias of `LineEdit`

class `qt_binder.raw_widgets.`**`Layout`**`(`**children*, ***kwds*`)`
Bases: *`qt_binder.binder.NChildren`*

Base class for all `QLayouts`.

qclass

construct`()`
Build the QLayout.

class `qt_binder.raw_widgets.`**`BoxLayout`**`(`**children*, ***kwds*`)`
Bases: *`qt_binder.raw_widgets.Layout`*

Base class for box layouts.

qclass

configure`()`

class `qt_binder.raw_widgets.`**`VBoxLayout`**`(`**children*, ***kwds*`)`
Bases: *`qt_binder.raw_widgets.BoxLayout`*

A vertical layout.

qclass

class `qt_binder.raw_widgets.`**`HBoxLayout`**`(`**children*, ***kwds*`)`
Bases: *`qt_binder.raw_widgets.BoxLayout`*

A horizontal layout.

qclass

class `qt_binder.raw_widgets.`**`StackedLayout`**`(`**children*, ***kwds*`)`
Bases: *`qt_binder.raw_widgets.Layout`*

A stacked layout.

qclass

configure`()`

class `qt_binder.raw_widgets.`**`FormLayout`**`(`**rows*, ***traits*`)`
Bases: *`qt_binder.raw_widgets.Layout`*

Children are (label, widget) pairs.

The label can be a `unicode` string or `None`. The last item can be a single `Binder` to take up the whole space.

qclass

child_binders = Property(List(Instance(Binder)))
The child Binder instances.

rows = List(Either(Tuple(Either(None, Unicode, Instance(Binder)), Instance(Binder)), Instance(Binder)))
The (label, widget) pairs.

configure()

---

class qt_binder.raw_widgets.**WithLayout**(*layout*, ***traits*)
Bases: *qt_binder.binder.Composite*

A dumb QWidget wrapper with a child Layout.

This is needed in some places where a true QWidget is needed instead of a QLayout.

qclass

configure()

---

class qt_binder.raw_widgets.**Splitter**(**children*, ***kwds*)
Bases: *qt_binder.binder.NChildren*

A splitter widget for arbitrary numbers of children.

qclass

construct()
Build the QLayout.

configure()

---

class qt_binder.raw_widgets.**ButtonGroup**(**button_ids*, ***traits*)
Bases: *qt_binder.binder.Binder*

A group of buttons.

This is a special Binder used in the button_groups= keyword to Bound. ButtonGroup is not a widget, so it does not get put into the widget hierarchy. It is given the ID strings of the button Binders that belong to the group.

qclass

button_ids = List(Either(Str, Tuple(Str, Int)))
List of Binder ID strings or (binder_id_str, qt_id_int)

add_buttons_from_context(*context*)
Pull out the required buttons from the context and add them.

2.4.5 qt_binder.type_registry

class qt_binder.type_registry.**TypeRegistry**
Bases: object

Register objects for types.

Each type maintains a stack of registered objects that can be pushed and popped.

push(*typ*, *obj*)
Push an object onto the stack for the given type.

Parameters

- **typ** (*type or '__module__:__name__' string for a type*) – The type the object corresponds to.
- **obj** (*object*) – The object to register.

push_abc(*typ*, *obj*)

Push an object onto the stack for the given ABC.

Parameters

- **typ** (*abc.ABCMeta*) – The ABC the object corresponds to.
- **obj** (*object*) – The object to register.

pop(*typ*)

Pop a registered object for the given type.

Parameters **typ** (*type or '__module__:__name__' string for a type*) – The type to look up.

Returns **obj** (*object*) – The last registered object for the type.

Raises

KeyError if the type is not registered.

lookup(*instance*)

Look up the registered object for the given instance.

Parameters **instance** (*object*) – An instance of a possibly registered type.

Returns **obj** (*object*) – The registered object for the type of the instance, one of the type's superclasses, or else one of the ABCs the type implements.

Raises

KeyError if the instance's type has not been registered.

lookup_by_type(*typ*)

Look up the registered object for a type.

typ : type

Returns **obj** (*object*) – The registered object for the type, one of its superclasses, or else one of the ABCs it implements.

Raises

KeyError if the type has not been registered.

lookup_all(*instance*)

Look up all the registered objects for the given instance.

Parameters **instance** (*object*) – An instance of a possibly registered type.

Returns **objs** (*list of objects*) – The list of registered objects for the instance. If the given instance is not registered, its superclasses are searched. If none of the superclasses are registered, search the possible ABCs.

Raises

KeyError if the instance's type has not been registered.

lookup_all_by_type(*typ*)

Look up all the registered objects for a type.

typ [type] The type to look up.

Returns **objs** (*list of objects*) – The list of registered objects for the type. If the given type is not registered, its superclasses are searched. If none of the superclasses are registered, search the possible ABCs.

Raises

`KeyError` if the type has not been registered.

---

class `qt_binder.type_registry.`**`LazyRegistry`**

Bases: *`qt_binder.type_registry.TypeRegistry`*

A type registry that will lazily import the registered objects.

Register '__module__:__name__' strings for the lazily imported objects. These will only be imported when the matching type is looked up. The module name must be a fully-qualified absolute name with all of the parent packages specified.

`lookup_by_type``(`*typ*`)`

Look up the registered object for a type.

2.4.6 `qt_binder.widgets`

Value-added wrappers for Qt widgets.

---

class `qt_binder.widgets.`**`TextField`**`(`**args*, ***traits*`)`

Bases: `qt_binder.raw_widgets.LineEdit`

Simple customization of a LineEdit.

The widget can be configured to update the model on every text change or only when Enter is pressed (or focus leaves). This emulates Traits UI's `TextEditor` `auto_set` and `enter_set` configurations.

If a validator is set, invalid text will cause the background to be red.

`value` **= Unicode(comparison_mode=NO_COMPARE)**

The value to sync with the model.

`mode` **= Enum('auto', 'enter')**

Whether the `value` updates on every keypress, or when Enter is pressed (or `focusOut`).

`valid` **= QtDynamicProperty(True)**

Whether or not the current value is valid, for the stylesheet.

`configure``()`

---

class `qt_binder.widgets.`**`EditableComboBox`**`(`**args*, ***traits*`)`

Bases: *`qt_binder.raw_widgets.ComboBox`*

ComboBox with an editable text field.

We do not do bidirectional synchronization of the value with the model since that is typically not required for these use cases.

`lineEdit_class`

alias of *`TextField`*

value = Any(Undefined, comparison_mode=NO_COMPARE)
The selected value.

values = List(Tuple(Any, Unicode))
(object, label) pairs.

same_as = Callable(operator.eq)
Function that is used to compare two objects in the values list for equality. Defaults to normal Python equality.

configure()

---

class qt_binder.widgets.**EnumDropDown**(**args*, ***traits*)
Bases: `qt_binder.raw_widgets.ComboBox`

Select from a set of preloaded choices.

value = Any(Undefined, comparison_mode=NO_COMPARE)
The selected value.

values = List(Tuple(Any, Unicode))
(object, label) pairs.

same_as = Callable(operator.eq)
Function that is used to compare two objects in the values list for equality. Defaults to normal Python equality.

---

class qt_binder.widgets.**UIFile**(*filename*, ***traits*)
Bases: `qt_binder.binder.Composite`

Load a layout from a Qt Designer `.ui` file.

Widgets and layouts with names that do not start with underscores will be added as traits to this `Binder`. The `binder_registry` will be consulted to find the raw `Binder` to use for each widget. This can be overridden for any named widget using the `overrides` trait.

qclass

filename = Str()
The .ui file with the layout.

overrides = Dict(Str, Instance(Binder))
Override binders for named widgets.

construct(**args*, ***kwds*)

---

class qt_binder.widgets.**BaseSlider**(**args*, ***traits*)
Bases: `qt_binder.raw_widgets.Slider`

Base class for the other sliders.

Mostly for interface-checking and common defaults.

value = Any(0)
The value to synch with the model.

range = Tuple(Any(0), Any(99))
The inclusive range.

qt_value = Rename('value')
The underlying Qt value.

orientation = Default(<DocMock.Unknown>)

class qt_binder.widgets.**IntSlider**(**args*, ***traits*)
Bases: *qt_binder.widgets.BaseSlider*

value = Int(0)
The value to synch with the model.

range = Tuple(Int(0), Int(99))
The inclusive range.

configure()

class qt_binder.widgets.**FloatSlider**(**args*, ***traits*)
Bases: *qt_binder.widgets.BaseSlider*

value = Float(0.0)
The value to synch with the model.

range = Tuple(Float(0.0), Float(1.0))
The inclusive range.

precision = Int(1000)
The number of steps in the range.

configure()

class qt_binder.widgets.**LogSlider**(**args*, ***traits*)
Bases: *qt_binder.widgets.FloatSlider*

range = Tuple(Float(0.01), Float(100.0))
The inclusive range.

class qt_binder.widgets.**RangeSlider**(**args*, ***traits*)
Bases: *qt_binder.binder.Composite*

A slider with labels and a text entry field.

The root widget is a `QWidget` with a new property `binder_class=RangeSlider`. Stylesheets can reference it using the selector:

```
*[binder_class="RangeSlider"] {...}
```

This can be useful for styling the child `QLabels` and `QLineEdit`, for example to make a series of `RangeSliders` align.

qclass

value = Any(0)
The value to synch with the model.

range = Tuple(Any(0), Any(99))
The inclusive range.

`label_format_func` = Callable(six.text_type)
The formatting function for the labels.

`field_format_func` = Callable(six.text_type)
The formatting function for the text field. This is used only when the slider is setting the value.

`field` = Instance(TextField, args=())
The field widget.

`slider` = Instance(BaseSlider, factory=IntSlider, args=())
The slider widget.

`construct`()

`configure`()

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index