qtypy Documentation

Release 1.0.0

Jérôme Laheurte

Apr 05, 2019

Contents:

CHAPTER 1

Application

1.1 Overview

qtypy.app.Application Application class

1.2 Reference

class qtypy.app.**Application**
Application class

setup_i18n(*locale_name=None*, *msg_path=None*)
Sets up i18n stuff and injects '_' in the builtin namespace.

CHAPTER 2

Settings

2.1 Overview

***qtypy.settings.Settings*:** Pythonic wrapper for QSettings

2.2 Example

```
from qtypy.settings import Settings

settings = Settings()
with settings.grouped('Geometry'):
    mainWindowGeometry = settings.value('MainWindow') if 'MainWindow' in settings␣
↪else None

for item in settings.array('RecentFiles'):
    self.addRecentFile(item.value('Path'))

...

with Settings() as settings: # So that sync() is called on exit
    settings.setValue('MainWindow', mainWindowGeometry)
    with settings.array('newarray') as items:
        for path in self.recentFiles():
            items.add(Path=path)
```

2.3 Reference

Pythonic wrapper for QSettings

class qtypy.settings.**Settings**

Settings class. You can also use this as a context manager so that *sync()* is called on exit.

items()

Yields all key/value pairs (recursively). Affected by the current group.

keys()

Yield all child keys. Affected by the current group.

keyValues()

Yield all child key/value pairs. Affected by the current group.

groups()

Yield all child groups. Affected by the current group.

grouped(*name*)

Context manager to enter a settings group. So instead of doing

```
settings.beginGroup('spam')
try:
    ...
finally:
    settings.endGroup()
```

you can do

```
with settings.grouped('spam'):
    ...
```

array(*name*)

Array support. This is used both to read a settings array (through iteration) and to write to it (using *with*). Example of reading:

```
for item in settings.array('myarray'):
    spam = item.value('spam')
    eggs = item.value('eggs')
```

is equivalent to:

```
for index in range(settings.beginReadArray('myarray')):
    spam = settings.value('spam')
    eggs = settings.value('eggs')
settings.endArray()
```

Example of writing:

```
with settings.array('myarray') as items:
    items.add(spam='spam', eggs='eggs')
    items.add(spam=42, eggs=13)
```

is equivalent to:

```
settings.beginWriteArray('myarray')
settings.setArrayIndex(0)
settings.setValue('spam', 'spam')
settings.setValue('eggs', 'eggs')
settings.setArrayIndex(1)
settings.setValue('spam', 42)
```

(continues on next page)

(continued from previous page)

```
settings.setValue('eggs', 13)
settings.endArray()
```

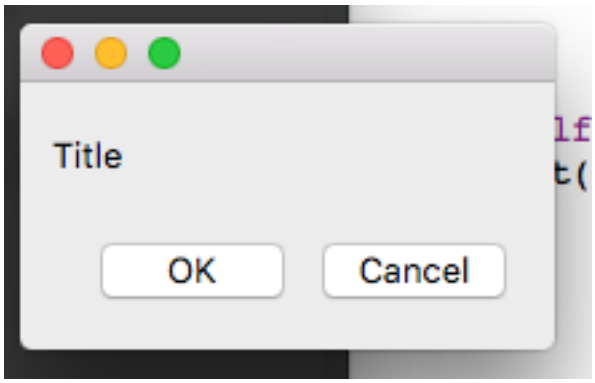
CHAPTER 3

Layout utilities

3.1 Overview

***qtypy.layout.LayoutBuilder*:** Stacking layouts with context managers.

3.2 Examples




```
#!/usr/bin/env python3

from PyQt5 import QtCore, QtWidgets

from qtypy.layout import LayoutBuilder


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        builder = LayoutBuilder(self)
        with builder.vbox() as vbox:
            vbox.addWidget(QtWidgets.QLabel('Title'))
            with builder.hbox() as hbox:
                hbox.addStretch(1)
                hbox.addWidget(QtWidgets.QPushButton('OK')).clicked.connect(self.onOK)
                hbox.addWidget(QtWidgets.QPushButton('Cancel')).clicked.connect(self.
↪onCancel)

        self.show()
        self.raise_()

    def onOK(self):
        print('== OK')

    def onCancel(self):
        print('== Cancel')


if __name__ == '__main__':
    app = QtWidgets.QApplication([])
    win = MainWindow()
    app.exec_()
```

3.3 Reference

Utilities for working with layouts.

class qtypy.layout.**LayoutBuilder**(*target*)

Layout builder. Use this to 'stack' layouts using context managers. For instance, instead of

```
l1 = QtWidgets.QHBoxLayout()
l1.addWidget(...)
l2 = QtWidgets.QVBoxLayout()
l2.addWidget(...)
l2.addLayout(l1)
self.setLayout(l2)
```

you can do

```
builder = LayoutBuilder(self)
with builder.vbox() as l2:
    l2.addWidget(...)
    with builder.hbox() as l1:
        l1.addWidget(...)
```

The builder class takes care of adding each layout to its parent (defined in the outer context manager), and adding the top-level layout to the target widget.

Methods that create a layout (hbox, vbox, etc) take additional positional and keyword arguments that will be passed to the parent's addLayout() method. Intermediate container widgets (when the top-level layout must be added to a QMainWindow for instance) are created automatically.

The returned layouts are actually proxies to actual layouts, with the *addWidget()* method returning the added widget. This allows call chaining without using an intermediate variable, for instance

```
hbox.addWidget(QtWidgets.QPushButton('OK')).clicked.connect(self.okSlot)
```

instead of

```
w = QtWidgets.QPushButton('OK')
hbox.addWidget(w)
w.clicked.connect(self.okSlot)
```

hbox(**args*, ***kwargs*)
Horizontal box.

vbox(**args*, ***kwargs*)
Vertical box.

stack(**args*, ***kwargs*)
Stack.

form(**args*, ***kwargs*)
Form.

split(**args*, ***kwargs*)
Splitter; this is not a layout strictly speaking but it will behave as one.

CHAPTER 4

Model/view helpers

4.1 Overview

qtypy.model.PythonListModel A class that behaves both as a Python list and a QAbstractListModel.

qtypy.model.PythonTreeModel A tree model based on PythonListModel which may contain other PythonListModel instances.

qtypy.widgets.view.ColumnedView A multi-columned view for PythonTreeModel.

- ***qtypy.widgets.view.Column*** Base class for columns
- ***qtypy.widgets.view.CheckableColumnMixin*** Mixin to make a column checkable
- ***qtypy.widgets.view.EditableColumnMixin*** Mixin to make a column editable
- ***qtypy.widgets.view.WidgetColumn*** Column that displays a widget, when all else fails

4.2 Examples

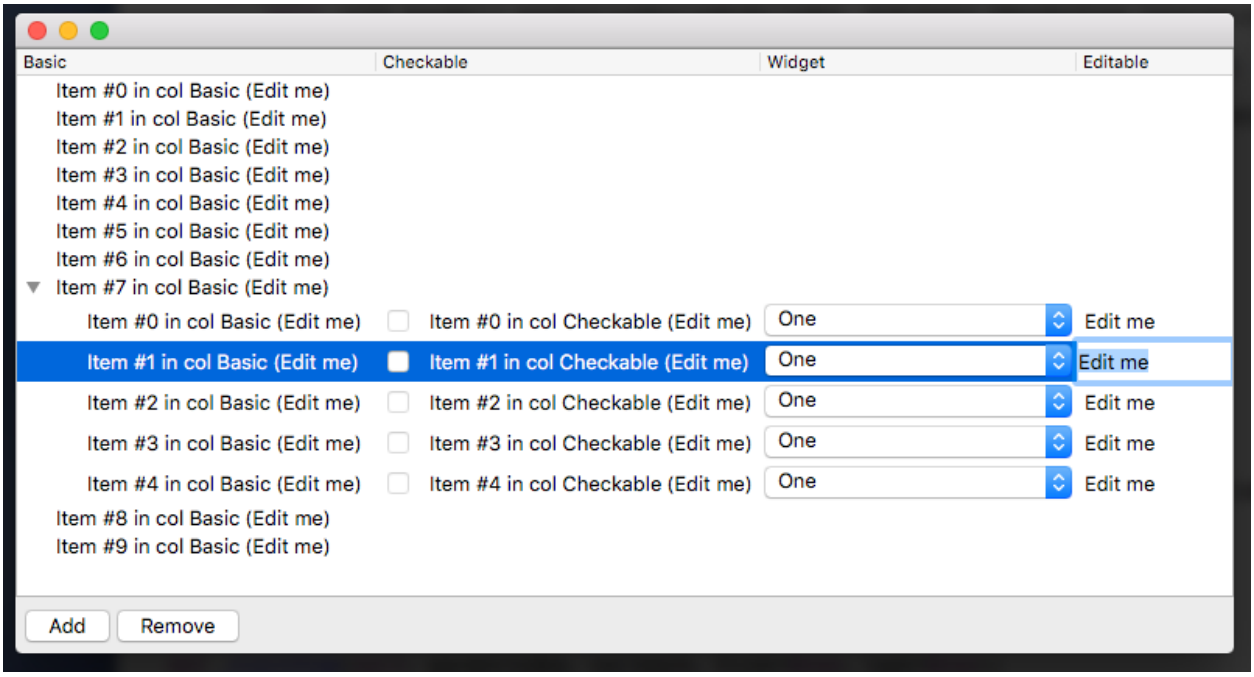
```
#!/usr/bin/env python3

from PyQt5 import QtCore, QtWidgets

from qtypy.settings import Settings
from qtypy.widgets.view import ColumnedView
from qtypy.model import PythonListModel
from qtypy.widgets.view import Column, WidgetColumn, CheckableColumnMixin,␣
↪EditableColumnMixin


class Item(PythonListModel):
    def __init__(self, n):
        super().__init__()
```

(continues on next page)



(continued from previous page)

```
        self.n = n
        self.checked = False
        self.text = 'Edit me'

        if n == 7:
            for i in range(5):
                self.append(Item(i))


class SimpleColumn(Column):
    def __init__(self, name):
        self._name = name
        super().__init__()

    def name(self):
        return self._name

    def id(self):
        # For saveState/restoreState
        return self._name

    def labelForItem(self, item):
        return 'Item #%d in col %s (%s)' % (item.n, self._name, item.text)


class CheckColumn(CheckableColumnMixin, SimpleColumn):
    def checkState(self, item):
        return QtCore.Qt.Checked if item.checked else QtCore.Qt.Unchecked

    def setCheckState(self, item, state):
        print('== Check state for item "%s": %d' % (self.labelForItem(item), state))
```

(continues on next page)

(continued from previous page)

```
        item.checked = state == QtCore.Qt.Checked

    def appliesToChildrenOf(self, parent):
        return parent is not None # Only for non-toplevel items


class EditColumn(EditableColumnMixin, SimpleColumn):
    def value(self, item):
        return item.text

    def setValue(self, item, value):
        print('== Set value for item "%s": %s' % (self.labelForItem(item), value))
        item.text = value

    def appliesToChildrenOf(self, parent):
        return parent is not None


class ComboColumn(WidgetColumn):
    def widgetFactory(self, item, parent):
        combo = QtWidgets.QComboBox(parent)
        combo.addItem('One')
        combo.addItem('Two')
        combo.addItem('Three')
        return combo

    def appliesToChildrenOf(self, parent):
        return parent is not None # Only for non-toplevel items


class CustomView(ColumnedView):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.setStandardButtons(self.BTN_ADD|self.BTN_DEL)
        self.addButtonClicked.connect(self._onAdd)
        self.delButtonClicked.connect(self._onDel)

    def _onAdd(self, selection):
        if selection:
            for item in selection:
                item.append(Item(42))
        else:
            self.model().append(Item(42))

    def _onDel(self, selection):
        for item in selection:
            parent = self.itemContainer(item)
            if parent is not None: # The parent itself may already have been removed
                parent.remove(item)

    def createContextMenu(self):
        # Column visibility menu
        menu = QtWidgets.QMenu(self)
        self.populateContextMenu(menu)
        return menu
```

(continues on next page)

(continued from previous page)

```
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        model = PythonListModel()
        view = CustomView(model=model, parent=self)

        view.addColumn(SimpleColumn('Basic')).setResizeMode(Column.Interactive)
        view.addColumn(CheckColumn('Checkable')).setResizeMode(Column.Contents)
        view.addColumn(ComboColumn('Widget')).setResizeMode(Column.Interactive)
        view.addColumn(EditColumn('Editable')).setResizeMode(Column.Stretch)
        self.setCentralWidget(view)

        for n in range(10):
            model.append(Item(n))

        with Settings().grouped('model') as settings:
            if 'state' in settings:
                view.restoreState(settings['state'])

        self.show()
        self.raise_()

    def closeEvent(self, event):
        with Settings().grouped('model') as settings:
            settings['state'] = self.centralWidget().saveState()
        event.accept()


if __name__ == '__main__':
    app = QtWidgets.QApplication([])
    app.setApplicationName('qtypy example')
    win = MainWindow()
    app.exec_()
```

4.3 Reference

4.3.1 Models

Classes for Qt's model/view programming, with a Pythonic touch.

class qtypy.model.**PythonListModel**(*parent=None*, *value=None*)

Python list-like object that implements QAbstractListModel. Most list operations like iterating, slicing, etc are supported except for

- reverse()
- sort()
- Slices with step

Items may define an *itemDataChanged()* signal, that is to be emitted whenever the item's data changes.

copy()

Returns a shallow copy of this object (same class, same content)

class qtypy.model.**PythonTreeModel**(*model=None*, *parent=None*)
This is an implementation of QAbstractItemModel based on a *PythonListModel*. Elements of the list model that are themselves instances of *PythonListModel* will have children.

itemContainer(*item*)
Looks up the item's container

itemParent(*item*)
Looks up the item's parent in the tree

4.3.2 Views

class qtypy.widgets.view.**ColumnedView**(*model*, *parent=None*)
A multi-column view for a PythonTreeModel or PythonListModel.

BTN_ADD = 1
Standard Add button

BTN_DEL = 2
Standard Remove button

selectionChanged = None
Signal emitted when the selection changes

addButtonClicked = None
Signal emitted when the "Add" button is clicked; current selection is passed as argument.

delButtonClicked = None
Signal emitted when the "Remove" button is clicked; current selection is passed as argument.

selection()
Returns the set of selected model objects

model()
Returns the PythonListModel of root elements

itemContainer(*item*)
Returns the PythonListModel that contains *item*

expandAll(*root=None*)
Expands all items

setStandardButtons(*buttons*)
Show standard buttons. The argument is an ORed combination of BTN_* values. The order and position of buttons will depend on the current platform's HIG.

standardButton(*type_*)
Returns the specified standard button

addButtonEnabled(*selection*)
Called to check if the 'Add' button should be enabled; return a boolean. The argument is a set of currently selected items (model objects). The default returns True.

delButtonEnabled(*selection*)
Called to check if the 'Remove' button should be enabled; return a boolean. The argument is a set of currently selected items (model objects). The default returns True if the selection is not empty.

saveState()
Returns a bytes object encapsulating the current state (column visibility, order, etc). You can save this in your settings and use it later to restore the state using *restoreState*.

restoreState(*state*)
Restore the state saved through *saveState*. If the versions do not match, nothing is done. Returns True if the state was restored.

addColumn(*column*, *visible=True*)
Append a column (instance of a *Column* subclass). Returns the column instance, for chaining.

createContextMenu()
This is called when then user right-clicks the view header. Return None for no context menu. You can use *populateContextMenu* to add actions to show/hide columns.

populateContextMenu(*menu*)
This appends to the menu as many checkable actions as there are columns, to show or hide them.

setEditTriggers(*triggers*)
Sets edit triggers for the underlying tree view

class qtypy.widgets.view.**Column**
Represents a column in a tree view. When you want to display data from a PythonTreeModel, use a *ColumnedView* and add instances of subclasses of this class to define columns.

Various mixins are available for common behavior (checkable, editable, etc).

Constant values for *setResizeMode*:

Variables

- **Interactive** – Column is user-resizable
- **Stretch** – Column takes all available space
- **Contents** – Column is resized according to contents

nameChanged = None
This signal should be emitted if the column name changes

id()
Return a persistent string identifier for this column; this is used by saveState/restoreState in ColumnedView.

name()
Return the column's name, or None.

appliesToChildrenOf(*parent*)
Return True if this column applies to children of *parent*

setVisible(*visible*)
Sets the current column visibility. This must be called *after* the column has been added to a view.

setResizeMode(*mode*)
Sets the resize mode. Possible values are either class attributes Interactive, Stretch or Contents, or an integer for a fixed size.

Note: by default all columns are user-resizable, except for the last one which is stretched.

---

show()
Short for setVisible(True)

hide()
Short for setVisible(False)

labelForItem(*item*)
Returns the text for this column for the given item. The default is to cast the item to *str*.

flags()
Called when the model needs to know flags associated with this column.

data(*item*, *role*)
Called when the model needs the item's data for this column. Mixins override this and provide specific methods (like *checkState()* in `CheckableColumnMixin`).

setData(*item*, *role*, *value*)
Called when the item's data for this column has been changed by the user and must be updated. Mixins override this and provide specific methods (like *setCheckState()* in `CheckableColumnMixin`).

class qtypy.widgets.view.**CheckableColumnMixin**
Mixin to make a column checkable.

checkState(*item*)
Override to return the check state for the item

setCheckState(*item*, *state*)
Override to update the item state

class qtypy.widgets.view.**EditableColumnMixin**
Mixin to make a column editable with the default editor.

value(*item*)
Override to return the item's edit value. The default is the item's label.

setValue(*item*, *value*)
Override to update the item's edit value.

class qtypy.widgets.view.**EditableTextColumn**(*column_name*, *attr_name*)
Concrete column class to display an editable text attribute

name()
Return the column's name, or None.

value(*item*)
Override to return the item's edit value. The default is the item's label.

setValue(*item*, *value*)
Override to update the item's edit value.

class qtypy.widgets.view.**CheckColumn**(*column_name*, *attr_name*)
Concrete column class to display a checkbox

name()
Return the column's name, or None.

labelForItem(*item*)
Returns the text for this column for the given item. The default is to cast the item to *str*.

checkState(*item*)
Override to return the check state for the item

setCheckState(*item*, *state*)
Override to update the item state

class qtypy.widgets.view.**WidgetColumn**(*column_name*)
A column that display a widget for each row.

name()
Return the column's name, or None.

labelForItem(*item*)
Returns the text for this column for the given item. The default is to cast the item to *str*.

widgetFactory(*item*, *parent*)
Widget factory. This is what you should overload.

Parameters

- **item** – The item
- **parent** – Parent for the new widget

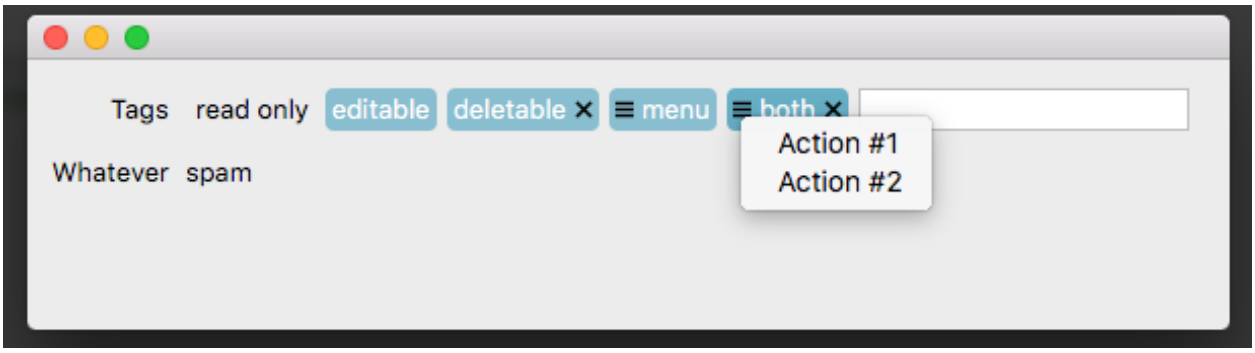
CHAPTER 5

Tag list widget

5.1 Overview

qtypy.widgets.taglist.TagList A tag list widget

5.2 Examples




```
#!/usr/bin/env python3

from PyQt5 import QtCore, QtWidgets

from qtypy.widgets.taglist import TagList
from qtypy.model import PythonListModel


class StringListModel(PythonListModel):
    def data(self, index, role):
        text = super().data(index, QtCore.Qt.UserRole)
```

(continues on next page)

(continued from previous page)

```
        if role in (QtCore.Qt.DisplayRole, QtCore.Qt.EditRole):
            return text

    def flags(self, index):
        flags = super().flags(index)
        if index.row() == 0:
            return int(flags)

        flags = int(flags | QtCore.Qt.ItemIsEditable)
        if index.row() == 1:
            pass
        if index.row() in (2, 4):
            flags |= TagList.ItemIsClosable
        if index.row() in (3, 4):
            flags |= TagList.ItemHasMenu
        return flags

    def setData(self, index, role, text):
        self[index.row()] = text


class MyTagList(TagList):
    def __init__(self, parent):
        super().__init__(StringListModel(value=['read only', 'editable', 'deletable',
↪'menu', 'both']), parent)

        cmpl = QtWidgets.QCompleter(['one', 'two', 'three', 'four'])
        self.setCompleter(cmpl)

    def addTag(self, text):
        self.model().append(text)

    def removeTag(self, index):
        self.model().pop(index)

    def popupMenu(self, index, pos):
        menu = QtWidgets.QMenu(self)
        menu.addSection(self.model()[index])
        menu.addSeparator()
        menu.addAction('Action #1')
        menu.addAction('Action #2')
        menu.popup(pos)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        container = QtWidgets.QWidget(self)
        layout = QtWidgets.QFormLayout()
        layout.addRow('Tags', MyTagList(self))
        layout.addRow('Whatever', QtWidgets.QLabel('spam'))
        layout.setFieldGrowthPolicy(layout.ExpandingFieldsGrow)
        container.setLayout(layout)
        self.setCentralWidget(container)

        self.resize(800, 600)
```

(continues on next page)

(continued from previous page)

```
        self.show()
        self.raise_()


if __name__ == '__main__':
    app = QtWidgets.QApplication([])
    win = MainWindow()
    app.exec_()
```

5.2.1 Classes

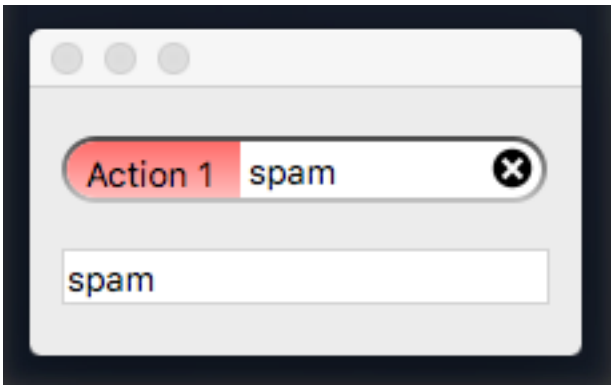
CHAPTER 6

Search widget

6.1 Overview

qtypy.widgets.search.SearchCtrl A search widget

6.2 Examples




```
#!/usr/bin/env python3

from PyQt5 import QtWidgets

from qtypy.widgets.search import SearchCtrl
from qtypy.layout import LayoutBuilder


class SearchWidget(QtWidgets.QWidget):
    def __init__(self, parent):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(parent)

        bld = LayoutBuilder(self)
        with bld.vbox() as layout:
            search = SearchCtrl(self)
            layout.addWidget(search)
            echo = QtWidgets.QLineEdit(self)
            layout.addWidget(echo)
            echo.setReadOnly(True)
            search.textChanged.connect(echo.setText)

        search.addAction(QtWidgets.QAction('Action 1', self))
        search.addAction(QtWidgets.QAction('Action 2', self), select=True)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        self.setCentralWidget(SearchWidget(self))
        self.show()
        self.raise_()


if __name__ == '__main__':
    app = QtWidgets.QApplication([])
    win = MainWindow()
    app.exec_()
```

6.2.1 Classes

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

L

M

N

P

Q

R

S

V

W