PyQt5 Tutorial Documentation

Release 1.0

Andrew Steele

March 01, 2016

CONTENTS

Author: Andrew Steele
Last updated: March 01, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Getting Started

Before starting with GUI programming in any language or using any toolkit, it is required to have a good understanding of the programming language in use. In the case of Python, it is important to know about variables and their types, using functions, and dealing with loops and if statements. It is suggested that the developer be capable of writing simple scripts and have some experience of using other modules.

This tutorial does not guide through the process of building an application from start to finish. It simply provides an overview of each widget in Qt, and shows how they work, combined with a simple example showcasing the widget basics.

1.2 License

1.3 Versioning

This tutorial was written on Ubuntu 14.10, with the examples developed and tested using Python 3.4.2 and Qt/PyQt 5.3.2. Although older versions may work for the most part, there may be some issues with missing methods, and bugs. Typically, the more up-to-date the software, the easier the development should be.

1.4 Contact

If you have any comments, or (constructive) criticism of the tutorial, feel free to contact me at andrew@andrewsteele.me.uk. Also feel free to submit changes via GitHub.

CHAPTER

TWO

HELLO WORLD

As is typical with any programming guide or tutorial, a "Hello, World!" example is required. This gives a basic example of creating a graphical window, and displaying some content in it.

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)
        self.setWindowTitle("Hello")

        layout = QGridLayout()
        self.setLayout(layout)

        label = QLabel("Hello, World!")
        layout.addWidget(label, 0, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `PushButton`

2.1 Stepping Through The Code

The first line is the hashbang (also known as crunchbang, shebang) which declares the Python interpreter version to use.

The import statements on the second and third lines allow us to import additional modules, including Qt.

The class statement defines our window and the type of object it will be, in this case QWidget. The `QWidget.__init__(self)` defines that the class is the QWindow object and allows setting of *Window* methods directly on the class.

The ninth line in the example defines the title of the Window, and is displayed on the titlebar if shown by your desktop environment/window manager.

Window object in Qt can only display one object at a time. To allow additional objects to be added, a container is used that can display multiple items. In this case, the *GridLayout* is used and subsequently assigned to the Window.

On line fourteen, the *Label* is constructed, and the parameter passed is the "Hello, World!" string which will be displayed. Line fifteen is then used to pack the label into the layout, with the `0, 0` indicating the position in the grid the top-left corner of the label will be attached.

Once the class has constructed itself, the Application object is constructed.

On line nineteen, the Window class is instantiated and then shown.

The Qt main loop is then executed inside the `sys.exit` statement, allowing the Python interpreter to exit when the main loop execution is ended.

CHAPTER

THREE

WINDOW

The Window is typically the base of every graphical application, and is used to display other widgets.

3.1 Constructor

Construction of the Window is done using:

```
window = QWindow()
```

3.2 Methods

The title of the Window, which is usually displayed by the Window Manager can be set using:

```
window.setTitle(title)
```

Window objects can also be minimised or maximised programatically using:

```
window.showMinimized()
window.showMaximized()
```

Alternatively, some applications will want a fullscreen mode:

```
window.showFullScreen()
```

If the window is set to minimised, maximised or fullscreen, it can be restored to a normal state by:

```
window.setNormal()
```

Minimum widths and heights are enforceable with:

```
window.setMinimumWidth(width)
window.setMaximumWidth(width)
window.setMinimumHeight(height)
window.setMaximumHeight(height)
```

A specific width and/or height can also be declared via:

```
window.setWidth(width)
window.setHeight(height)
```

3.3 Example

Below is an example of a Window:

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

from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
import sys

class Window(QWindow):
    def __init__(self):
        QWindow.__init__(self)
        self.setTitle("Window")
        self.resize(400, 300)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Window`

CHAPTER

FOUR

BOXLAYOUT

The BoxLayout is similar to the *GridLayout*, however it only supports a single row or column of widgets depending on the orientation. It does however dynamically size to the number of widgets it is to contain.

4.1 Constructor

The constructor for the BoxLayout is:

```
boxlayout = QBoxLayout()
```

4.2 Methods

Widgets are inserted into the BoxLayout with the methods:

```
boxlayout.addWidget(widget, stretch, alignment)
boxlayout.insertWidget(index, widget, stretch, alignment)
```

An *index* value in the `.insertWidget` method indicates the location at which the child widget should be placed. The *widget* parameter is the child widget which is to be added to the BoxLayout. The *stretch* value should be set to an integer indicating the factor at which the child widget stretches to fill the space. Finally, the *alignment* value can be set to one of the following:

- `Qt.AlignmentLeft`
- `Qt.AlignmentRight`
- `Qt.AlignmentHCenter`
- `Qt.AlignmentJustify`

Layout objects are added to the BoxLayout via alternative methods:

```
boxlayout.addLayout(layout, stretch)
boxlayout.insertLayout(index, layout, stretch)
```

The pixel spacing between each child widget defaults to zero, however this is configurable with:

```
boxlayout.setSpacing(spacing)
```

Spacing can be added as with a normal widget by:

```
boxlayout.addSpacing(spacing)
boxlayout.insertSpacing(index, spacing)
```

The *spacing* value indicates the number of pixels spacing to be displayed. The `.insertSpacing()` method also takes an *index* indicating the location at which the spacing should be inserted.

The direction of the BoxLayout is settable with the method:

```
boxlayout.setDirection(direction)
```

The *direction* parameter should be set to one of the following:

- `QBoxLayout.LeftToRight`
- `QBoxLayout.RightToLeft`
- `QBoxLayout.TopToBottom`
- `QBoxLayout.BottomToTop`

4.3 Example

Below is an example of a BoxLayout:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QBoxLayout(QBoxLayout.LeftToRight)
        self.setLayout(layout)

        label = QLabel("Label 1")
        layout.addWidget(label, 0)
        label = QLabel("Label 2")
        layout.addWidget(label, 0)

        layout2 = QBoxLayout(QBoxLayout.TopToBottom)
        layout.addLayout(layout2)

        label = QLabel("Label 3")
        layout2.addWidget(label, 0)
        label = QLabel("Label 4")
        layout2.addWidget(label, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `BoxLayout`

CHAPTER

FIVE

GRIDLAYOUT

The GridLayout widget provides a container which allows widgets to be laid out in a dynamically sized grid.

5.1 Constructor

The constructor for the GridLayout is:

```
gridlayout = QGridLayout()
```

5.2 Methods

Items are added to the GridLayout using:

```
gridlayout.addWidget(widget)
gridlayout.addWidget(widget, row, column)
gridlayout.addWidget(widget, row, column, rowspan, columnspan, alignment)
```

The *widget* parameter indicates the widget which is to be added to the GridLayout at *row* and *column*. The row and column values work on a coordinate-like system, with 0 and 0 indicating top-left. The *rowspan* and *columnspan* values indicate how many rows or columns the widget should span. Finally, the *alignment* parameter should be set to one of the following:

- `Qt.AlignmentLeft`
- `Qt.AlignmentRight`
- `Qt.AlignmentHCenter`
- `Qt.AlignmentJustify`

A layout is added to the GridLayout using alternative methods:

```
gridlayout.addLayout(widget)
gridlayout.addLayout(widget, row, column)
gridlayout.addLayout(widget, row, column, rowspan, columnspan, alignment)
```

Retrieving the item at a given position is done with the method:

```
gridlayout.itemAtPosition(row, column)
```

There is no spacing between rows and columns by default. This can be adjusted via:

```
gridlayout.setSpacing(spacing)
```

Alternatively, vertical and horizontal spacing can be specified separately using:

```
gridlayout.setHorizontalSpacing(spacing)
gridlayout.setVerticalSpacing(spacing)
```

The *spacing* parameter should be set to an integer number indicating the number of pixels spacing which should be displayed.

The number of rows and columns can be obtained from the container with:

```
gridlayout.rowCount()
gridlayout.columnCount()
```

5.3 Example

Below is an example of a GridLayout:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        label = QLabel("Label (0, 0)")
        layout.addWidget(label, 0, 0)
        label = QLabel("Label (0, 1)")
        layout.addWidget(label, 0, 1)
        label = QLabel("Label (1, 0) spanning 2 columns")
        layout.addWidget(label, 1, 0, 1, 2)
        label = QLabel("Label (1, 0) spanning 2 rows")
        layout.addWidget(label, 0, 2, 2, 1)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `GridLayout`

CHAPTER

SIX

LABEL

The Label widget is used to display text to the user. This can be anything from one-word labels indicating the purpose of another widget, to single sentences, to multi-line, multi-paragraph blocks of text.

6.1 Constructor

Label widgets are constructed via the constructor:

```
label = QLabel(text)
```

The *text* parameter can either be left-out, with the text optionally being specified later, or defined at construction time.

6.2 Methods

To set or change the text after construction, call:

```
label.setText(text)
```

Text can also be retrieved from the Label via:

```
label.text()
```

Alignment defaults for the Label is to position text to the left of the label, and central vertically. This can be customised:

```
label.setAlignment(alignment)
```

The *alignment* parameter specifies where to place the text both horizontally and vertically. The horizontal constants are:

- `Qt.AlignLeft`
- `Qt.AlignHCenter`
- `Qt.AlignRight`
- `Qt.AlignJustify`

To set the vertical alignment position:

- `Qt.AlignTop`
- `Qt.AlignVCenter`
- `Qt.AlignBottom`
- `Qt.AlignBaseline`

If both horizontal and vertical alignments are needed, the constants should be separated by a pipe |.

The Label widget also allows wrapping of text if there are multiple lines. This can be enabled using the method:

```
label.setWordWrap(word_wrap)
```

When *word_wrap* is set to `True`, the text will be wrapped into the space allocated for the widget.

The margin size on a Label is zero initially. Custom margin settings are allowed by specifying the size in pixels:

```
label.setMargin(margin)
```

Indents can also be applied to the Label text by specifying the indent amount in pixels:

```
label.setIndent(indent)
```

Mnemonic keyboard shortcuts are an important part of accessibility and speed when using an application. They are identified by the presence of an underscore beneath a letter in the label. Some widgets however can not display a mnemonic character, so a Label can be paired with the other widget. This allows focus to be transferred to the other widget from the Label when the shortcut key is used.

label.setBuddy(widget)

The *widget* parameter is the name of the widget to be paired with the Label.

6.3 Example

Below is an example of a Label:

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

from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        label = QLabel("The Story of Dale")
        layout.addWidget(label, 0, 0)

        label = QLabel("Few people could understand Dale's motivation. It wasn't something that was e
        label.setWordWrap(True)
        layout.addWidget(label, 0, 1)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Label`

CHAPTER

SEVEN

PUSHBUTTON

The PushButton is often used to get the program to do something with the user simply having to press a button. This could be starting a download or deleting a file.

7.1 Constructor

The PushButton is constructed using:

```
pushbutton = QPushButton(label)
```

The *label* string can be left out if not required, or set to the text which should be shown on top of the button.

7.2 Methods

The label displayed on the button can be changed after widget construction by:

```
pushbutton.setText(label)
```

By default, the button is shown with a well-defined border making it appeared raised up from the surface of the window beneath. It is possible however to give the button a flat appearance via:

```
pushbutton.setFlat(flat)
```

When *flat* is set to `True`, the button does not appear raised.

To check whether a button has been set to flat or not, call:

```
pushbutton.isFlat()
```

Button widgets can also be used to display a dropdown menu rather than simply being clickable. The menu is associated using:

```
pusbutton.setMenu(menu)
```

The *menu* parameter should be set to the name of a *Menu* widget.

7.3 Signals

One of the common functions of a button is to be clicked by the user, and perform an associated action. This is done by connecting the clicked signal of the button to the appropriate function:

```
pushbutton.clicked.connect(button_clicked_function)
```

7.4 Example

Below is an example of a PushButton:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        button = QPushButton("Click Me")
        button.clicked.connect(self.on_button_clicked)
        layout.addWidget(button, 0, 0)

    def on_button_clicked(self):
        print("The button was pressed!")

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `PushButton`

CHAPTER

EIGHT

RADIOBUTTON

The RadioButton is a toggable button, which is typically used in conjunction with other RadioButton's with only one of the buttons able to be selected at any one time.

If multiple items should be set at one time, a *CheckBox* or *PushButton* operating in toggle-mode can be used.

8.1 Constructor

The constructor used for building the RadioButton is:

```
radiobutton = QRadioButton(label)
```

8.2 Methods

Text can be changed within the RadioButton via:

```
radiobutton.setText(label)
```

The text can also be retrieved from the RadioButton by using the method:

```
radiobutton.text()
```

To set a RadioButton to be checked, use:

```
radiobutton.setChecked(checked)
```

When *checked* is set to `True`, the defined RadioButton will be active.

Determining whether the RadioButton is active or not is done by:

```
radiobutton.isChecked()
```

By default, all RadioButton widgets within the window will be assigned to the same group. This will cause problems if there are multiple batches of buttons which have different intents. To resolve this issue, read about the *ButtonGroup* object.

An icon can also be applied to the RadioButton if required:

```
radiobutton.setIcon(icon)
```

8.3 Example

Below is an example of a RadioButton:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        radiobutton = QRadioButton("Brazil")
        radiobutton.setChecked(True)
        radiobutton.country = "Brazil"
        radiobutton.toggled.connect(self.on_radio_button_toggled)
        layout.addWidget(radiobutton, 0, 0)

        radiobutton = QRadioButton("Argentina")
        radiobutton.country = "Argentina"
        radiobutton.toggled.connect(self.on_radio_button_toggled)
        layout.addWidget(radiobutton, 0, 1)

        radiobutton = QRadioButton("Ecuador")
        radiobutton.country = "Ecuador"
        radiobutton.toggled.connect(self.on_radio_button_toggled)
        layout.addWidget(radiobutton, 0, 2)

    def on_radio_button_toggled(self):
        radiobutton = self.sender()

        if radiobutton.isChecked():
            print("Selected country is %s" % (radiobutton.country))

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `RadioButton`

CHAPTER

NINE

CHECKBOX

A CheckBox provides a checked or unchecked state, indicated via a tick in a box. These are commonly used to indicate when a feature is enabled.

9.1 Constructor

Constructing the CheckBox is done with the following statement:

```
checkbox = QCheckBox(text)
```

The *text* parameter is optional. When included, the CheckBox will be displayed with an associated textual label, typically indicating the purpose of the option.

9.2 Methods

The text associated with the CheckBox can be set after construction by calling:

```
checkbox.setText(text)
```

Adjusting the CheckBox state programmatically is done with the method:

```
checkbox.setChecked(checked)
```

When *checked* is set to `True`, the CheckBox will contain a ticket in the box.

To get the state of the CheckBox, use:

```
checkbox.isChecked()
```

A tick in the CheckBox will return `True` from the method, while `False` is returned when the CheckBox is unchecked.

By default, a CheckBox can be true or false. A third (tri-state) is possible, and is enabled using:

```
checkbox.setTristate(tristate)
```

When *tristate* is set to `True`, the CheckBox will display a line through the indicator box. The tri-state is commonly used to show a mismatch between other set options.

To check whether a CheckBox is enabled for tri-state, use the method:

```
checkbox.isTristate()
```

The `.ischecked()` method can only be used for CheckBox widgets which do not use the tri-state setting. To obtain the state when tri-state is being used, call:

```
checkbox.checkState()
```

A tri-state enabled CheckBox status can be set using the method:

```
checkbox.setCheckState(state)
```

The state should be set to one of the following values:

- `Qt.Unchecked` - the item is not checked.
- `Qt.PartiallyChecked` the item is in a partial checked state.
- `Qt.Checked` - the item is checked.

9.3 Example

Below is an example of a CheckBox:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.checkbox1 = QCheckBox("Kestrel")
        self.checkbox1.setChecked(True)
        self.checkbox1.toggled.connect(self.checkbox_toggled)
        layout.addWidget(self.checkbox1, 0, 0)

        self.checkbox2 = QCheckBox("Sparrowhawk")
        self.checkbox2.toggled.connect(self.checkbox_toggled)
        layout.addWidget(self.checkbox2, 1, 0)

        self.checkbox3 = QCheckBox("Hobby")
        self.checkbox3.toggled.connect(self.checkbox_toggled)
        layout.addWidget(self.checkbox3, 2, 0)

    def checkbox_toggled(self):
        selected = []

        if self.checkbox1.isChecked():
            selected.append("Kestrel")

        if self.checkbox2.isChecked():
            selected.append("Sparrowhawk")

        if self.checkbox3.isChecked():
            selected.append("Hobby")

        print("Selected: %s" % (" ".join(selected)))

app = QApplication(sys.argv)
```

```
screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: CheckBox

CHAPTER

TEN

TOOLTIP

ToolTip widgets are attached to other widgets and appear when the user hovers over the widget, displaying hints about the purpose of the widget.

10.1 Methods

10.2 Example

Below is an example of a ToolTip:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        button = QPushButton("Simple ToolTip")
        button.setToolTip("This ToolTip simply displays text.")
        layout.addWidget(button, 0, 0)

        button = QPushButton("Formatted ToolTip")
        button.setToolTip("<b>Formatted text</b> can also be displayed.")
        layout.addWidget(button, 1, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `ToolTip`

CHAPTER

ELEVEN

WHATSTHIS

The "WhatsThis" class provides a description of the purpose of any widget. Although similar to a tooltip, the WhatsThis description is longer and more detailed, but generally they provide less information than a help window.

11.1 Constructor

The WhatsThis object is not constructed separately, but is able to be attached to most widgets or actions by the method:

```
widget.setWhatsThis(text)
```

The *text* string should be defined to explain the purpose of the widget.

11.2 Example

Below is an example of a WhatsThis object:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        label = QLabel("Focus ComboBox and press SHIFT+F1")
        layout.addWidget(label)

        self.combobox = QComboBox()
        self.combobox.setWhatsThis("This is a 'WhatsThis' object description.")
        layout.addWidget(self.combobox)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `WhatsThis`

CHAPTER

TWELVE

LINEEDIT

The LineEdit widget is a one-line text entry widget used to receive textual input from the user. Example use cases include the user entering their name or their password.

12.1 Methods

By default, the LineEdit has no text displayed within the widget. In some cases it may be useful to have a prepopulated string which can be set with:

```
lineedit.setText(text)
lineedit.insert(text)
```

Both of the methods overwrite any existing text.

Text is also retrieved from the widget by:

```
lineedit.text()
```

Another useful feature is to show placeholder text in the LineEdit, which indicates the widgets purpose:

```
lineedit.setPlaceholderText(text)
```

To prevent the user from modifying the content of the LineEdit, call:

```
lineedit.setReadOnly(read_only)
```

When *read_only* is set to `True`, the widget will not allow its content to be modified.

The default setting of the LineEdit is to allow 32767 characters to be entered into the field. This can be limited by:

```
lineedit.setMaxLength(length)
```

A *Completer* can be added to the LineEdit using the method:

```
lineedit.setCompleter(completer)
```

12.2 Signals

If the user pressed the `Enter` or `Return` buttons after editing the text, the LineEdit can be made to run a function:

```
lineedit.returnPressed.connect(return_pressed_function)
```

Alternatively, it may be useful to run on a function after each change made:

```
lineedit.textChanged.connect(text_changed_function)
```

12.3 Example

Below is an example of a LineEdit:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.lineedit = QLineEdit()
        self.lineedit.returnPressed.connect(self.return_pressed)
        layout.addWidget(self.lineedit, 0, 0)

    def return_pressed(self):
        print(self.lineedit.text())

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `LineEdit`

CHAPTER

THIRTEEN

BUTTONGROUP

A ButtonGroup is an invisible object used to group buttons. It is typically used with *RadioButton* widgets to prevent them interacting with other RadioButton's not intended to be in the same group.

13.1 Constructor

A ButtonGroup is constructed with the call:

```
buttongroup = QButtonGroup()
```

13.2 Methods

A button is added to the group with the method:

```
buttongroup.addButton(button, id)
```

The *button* parameter indicated the button to be added into the ButtonGroup. The *id* value can be left if not required, in which case it will be assigned a negative value. If it is specified, the value should be positive. The value allows a button to be identified within the grouping.

To remove a button from the group, use the call:

```
buttongroup.removeButton(button)
```

A list of all the buttons associated with the ButtonGroup can be made via:

```
buttongroup.buttons()
```

Using the id property when adding the buttons, a button object can be retrieved for a given id with:

```
buttongroup.button(id)
```

On the reverse, an id for a given button can also be fetched:

```
buttongroup.id(button)
```

If the id is to be specified after the button has been added to the ButtonGroup, call:

```
buttongroup.setId(button, id)
```

To enforce that only one button in the group can be selected at a time, use:

```
buttongroup.setExclusive(exclusive)
```

If the ButtonGroup contains buttons which can be in the checked state, the active button can be found with:

```
buttongroup.checkedButton()
```

13.3 Signals

The available ButtonGroup signals are:

```
buttonClicked(button)
buttonClicked(id)
buttonPressed(button)
buttonPressed(id)
buttonReleased(button)
buttonReleased(id)
buttonToggled(button)
buttonToggled(id)
```

Either the button object or id value can be connected, which will be actioned when the group member is clicked, pressed, released, or toggled.

13.4 Example

Below is an example of a ButtonGroup:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.buttongroup = QButtonGroup()
        self.buttongroup.setExclusive(False)
        self.buttongroup.buttonClicked[int].connect(self.on_button_clicked)

        button = QPushButton("Button 1")
        self.buttongroup.addButton(button, 1)
        layout.addWidget(button)

        button = QPushButton("Button 2")
        self.buttongroup.addButton(button, 2)
        layout.addWidget(button)

    def on_button_clicked(self, id):
        for button in self.buttongroup.buttons():
            if button is self.buttongroup.button(id):
                print("%s was clicked!" % (button.text()))
```

```
app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: ButtonGroup

CHAPTER

FOURTEEN

GROUPBOX

The GroupBox provides a tidy way to group items, with the container featuring a title label and bordering frame.

It should be noted that the GroupBox can only contain one widget itself, with the intention of other containers such as a *BoxLayout*.

14.1 Constructor

The constructor for a GroupBox is:

```
groupbox = QGroupBox(title)
```

The *title* parameter should be set with the string of text to display.

14.2 Methods

The title applied to the GroupBox can be set using:

```
groupbox.setTitle(title)
```

A widget is added to the GroupBox with:

```
groupbox.setLayout(child)
```

The alignment of children within the GroupBox is settable via:

```
groupbox.setAlignment(alignment)
```

By default, the alignment is set to the left-edge, however it can be customised with the *alignment* value being set to one of the following:

- `Qt.AlignLeft`
- `Qt.AlignRight`
- `Qt.AlignHCenter`

The GroupBox can be made checkable if required. This permits all child *CheckBox* or *RadioButton* widgets to be made sensitive or insenstive. This is set via:

```
groupbox.setCheckable(checkable)
```

The checked state of the GroupBox can be obtained using:

```
groupbox.isChecked()
```

Programatically setting the checked state of the GroupBox can be done using:

```
groupbox.setChecked(checked)
```

When *checked* is set to `True`, the GroupBox checkbox will contain a tick. Setting to `False` will removed the tick.

14.3 Example

Below is an example of a GroupBox:

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

from PyQt5.QtWidgets import *
import sys

class GroupBox(QWidget):
    def __init__(self):
        QWidget.__init__(self)
        self.setWindowTitle("GroupBox")

        layout = QGridLayout()
        self.setLayout(layout)

        groupbox = QGroupBox("GroupBox Example")
        groupbox.setCheckable(True)
        layout.addWidget(groupbox)

        vbox = QVBoxLayout()
        groupbox.setLayout(vbox)

        radiobutton = QRadioButton("RadioButton 1")
        radiobutton.setChecked(True)
        vbox.addWidget(radiobutton)
        radiobutton = QRadioButton("RadioButton 2")
        vbox.addWidget(radiobutton)

app = QApplication(sys.argv)

screen = GroupBox()
screen.show()

sys.exit(app.exec_())
```

Download: `GroupBox`

CHAPTER

FIFTEEN

SIZEGRIP

The SizeGrip widget provides a way to resize a parent *Window*. It commonly appears as a triangle in the bottom right corner of the window and allows the user to increase or decreate the window width and height.

15.1 Constructor

The SizeGrip is constructable with the call:

```
sizegrip = QSizeGrip(parent)
```

The *parent* parameter should be set to the parent widget to be assigned the SizeGrip.

15.2 Methods

To configure the visibility of the SizeGrip use:

```
sizegrip.setVisible(visible)
```

15.3 Example

Below is an example of a SizeGrip:

Download: `SizeGrip`

CHAPTER

SIXTEEN

SPLITTER

The Splitter is an organiser class widget, which provides a way to insert child items which can then be given varying amounts of space. The amount of space allowed is adjusted by the user using a handle on the Splitter.

The widget is commonly seen in File Managers and Web Browsers where the main content may also need to share space with a sidepanel such as a tree view or bookmark list.

16.1 Constructor

The Splitter can be constructed with:

```
splitter = QSplitter()
```

16.2 Methods

Child widgets can be added to the Splitter with the methods:

```
splitter.addWidget(widget)
splitter.insertWidget(index, widget)
```

The *widget* parameter is the name of the child widget to be inserted. The *index* value of the `.insertWidget()` method specifies the position to insert the widget at. The `.addWidget()` method adds items to the Splitter in the order the code is executed.

By default, the Splitter takes on a horizontal orientation. This can be changed with:

```
splitter.setOrientation(orientation)
```

The *orientation* value should be set to one of the following:

- `Qt.Horizontal`
- `Qt.Vertical`

In some cases, it may be useful to retrieve the widget for a given index, or the index for a given widget. This can be done using the methods:

```
splitter.widget(index)
splitter.indexOf(widget)
```

The number of widgets being held by the Splitter can also be found by:

```
splitter.count()
```

The width of the handle in pixels can be retrieved using:

```
splitter.handleWidth(width)
```

It can also be defined using:

```
splitter.setHandleWidth(width)
```

The *width* parameter again should be specified in pixels.

16.3 Example

Below is an example of a Splitter:

Download: `Splitter`

CHAPTER

SEVENTEEN

FRAME

The Frame container provides a grouping box with an associated title. Typically, widgets contained within the Frame are related to a particular function.

17.1 Constructor

The Frame is constructed using:

```
frame = QFrame()
```

17.2 Methods

The line width of the Frame can be set in pixels using:

```
frame.setLineWidth(width)
```

By default, the width of the line is `1`.

The Frame can take on three appearances; plain, raised, or sunken. This is configurable via:

```
frame.setFrameShadow(shadow)
```

The default appearance is plain. The *shadow* can be set however to one of the following:

- `QFrame::Plain`
- `QFrame::Raised`
- `QFrame::Sunken`

The shape of the frame can be set via:

```
frame.setFrameShape(shape)
```

The *shape* parameter should be set to one of the following:

- `QFrame::NoFrame` - draw no frame around the contents.
- `QFrame::Box` - draw a box around the contents.
- `QFrame::Panel` - draw a panel to make the content appear raised or sunken.
- `QFrame::StyledPanel` - draw a raised or sunken rectangular panel dependent on the interface style.
- `QFrame::HLine` - draw a horizontal line as a separator.

- `QFrame::VLine` - draw a vertical line as a separator.
- `QFrame::WinPanel` - draw a rectangular panel, raised or sunken, similar to those found in Windows 2000.

17.3 Example

Below is an example of a Frame:

Download: `Frame`

CHAPTER

EIGHTEEN

SLIDER

A Slider provides a way to adjust a numerical value by moving a slide along a run to change the output value. It is commonly seen when adjusting the volume of a speaker, or the brightness of a screen.

18.1 Constructor

Slider widgets are constructed using:

```
slider = QSlider(orientation)
```

By default, the Slider is oriented vertically with the slider object moving from top to bottom. The *orientation* parameter is optional, by can be set to `Qt.Vertical` or `Qt.Horizontal`.

18.2 Methods

The orientation can also be changed after construction with:

```
slider.setOrientation(orientation)
```

By default the slider ranges between 0 and 99. Custom minimum and maximum values can be defined:

```
slider.setMinimum(value)
slider.setMaximum(value)
```

If attempting to set a value on the slider which falls outside the minimum and maximum values, the value will be adjusted so that it falls in the range.

A value can be set onto the Slider using:

```
slider.setValue(value)
```

The Slider emits a signal that the value has changed whenever the user stops sliding and releases the mouse. In some cases, the requirement may be to emit a changed signal whenever the Slider moves. This can be done with:

```
slider.setTracking(tracking)
```

If *tracking* is set to `True`, the Slider will call the associated update function repeatedly when moving.

Ticks can be added to the Slider scale at set positions to ease the user in viewing where on the scale the marker is. The method for this is:

```
slider.setTickInterval(interval)
```

The *interval* value should be a number, which indicates the gap between each tick.

The position of the ticks can be configured via:

```
slider.setTickPosition(position)
```

The *position* value should be set to one of:

- `QSlider.NoTicks` - do not draw tick marks.
- `QSlider.TicksBothSides` - draw ticks on both sides of the scale.
- `QSlider.TicksAbove` - draw ticks above the horizontal slider.
- `QSlider.TicksBelow` - draw ticks below the horizontal slider.
- `QSlider.TicksLeft` - draw ticks to the left of the vertical slider.
- `QSlider.TicksRight` - draw ticks to the right of the vertical slider.

18.3 Example

Below is an example of a Slider:

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

from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        slider = QSlider(Qt.Horizontal)
        slider.setValue(4)
        layout.addWidget(slider, 0, 0)

        slider = QSlider(Qt.Vertical)
        slider.setValue(4)
        layout.addWidget(slider, 0, 1)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Slider`

CHAPTER

NINETEEN

SCROLLBAR

A ScrollBar provides a way to move horizontally or vertically within a frame where the content is to large to fit. The ScrollBar typically includes a bar with arrows buttons to move the view. A bar is also provided within to drag-and-drop into a new position.

19.1 Constructor

The ScrollBar is constructed using the call:

```
scrollbar = QScrollBar()
```

The orientation can also be defined at construction time via:

```
scrollbar = QScrollBar(orientation)
```

The *orientation* parameter should be set to one of the following:

- `Qt.Horizontal`
- `Qt.Vertical`

19.2 Example

Below is an example of a ScrollBar:

Download: `ScrollBar`

CHAPTER

TWENTY

SCROLLAREA

A ScrollArea widget provides a container for another widget to be placed, providing scrolling in both vertical and horizontal directions when the child is larger than the space allocated.

The ScrollArea automatically provides *ScrollBar* objects and is preferred in most cases when scrolling must be provided.

20.1 Constructor

Construction of the ScrollArea is made using:

```
scrollarea = QScrollArea()
```

20.2 Methods

Widgets are added to the ScrollArea container using:

```
scrollarea.setWidget(widget)
```

The widget assigned to the ScrollArea can be retrieved with:

```
scrollarea.widget()
```

The added widget can be positioned within the area via:

```
scrollarea.setAlignment(alignment)
```

Set the *alignment* value to one of the following:

- `Qt.AlignLeft`
- `Qt.AlignRight`
- `Qt.AlignTop`
- `Qt.AlignBottom`
- `Qt.AlignHCenter`
- `Qt.AlignVCenter`

The child widget can be resized within the ScrollArea via:

```
scrollarea.setWidgetResizable(resizable)
```

When *resizable* is set to `True`, the ScrollArea automatically resizes the widget to try and avoid scroll bars and take advantage of extra space. If set to `False`, the default widget size is honoured.

20.3 Example

Below is an example of a ScrollArea:

Download: `ScrollArea`

CHAPTER

TWENTYONE

DIAL

The Dial widget provides a range object which takes the form of a control knob. Its design is similar to a volume knob on a music system, with the turning of the dial outputting different numbers within a defined range.

Note: The Dial widget may change appearance based on the platform in use, however the functionality remains the same.

21.1 Constructor

The Dial widget is created by defining:

```
dial = QDial()
```

21.2 Methods

The minimum and maximum values of the Dial are set by:

```
dial.setMinimum(minimum)
dial.setMaximum(maximum)
```

To set the value of the Dial programmatically, call:

```
dial.setValue(value)
```

Retrieving the value set on the Dial is done using:

```
dial.value()
```

The minimum and maximum values are also retrievable with the method:

```
dial.minimum()
dial.maximum()
```

By default, the Dial will wrap so that dragging from the highest number will reset the Dial back to the lowest. This can be configured with:

```
dial.setWrapping(wrapping)
```

When *wrapping* is set to `False`, the user will need to drag the Dial all the way back around from the highest to lowest point.

A notch target can be defined. This holds the number of pixels which the Dial attempts to place between notches, with a default of 3.7 pixels. This can be modified by the method:

```
dial.setNotchTarget(target)
```

Notches can also be toggled visible or invisible with:

```
dial.setNotchesVisible(visible)
```

21.3 Example

Below is an example of an Dial:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.dial = QDial()
        self.dial.setMinimum(0)
        self.dial.setMaximum(100)
        self.dial.setValue(30)
        self.dial.valueChanged.connect(self.slider_changed)
        layout.addWidget(self.dial)

    def slider_changed(self):
        print("Current dial value: %i" % (self.dial.value()))

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Dial`

CHAPTER

TWENTYTWO

SPINBOX

The SpinBox widget provides a way to enter numerical data. Thw widget provides integrated adjustment buttons which allow the user to adjust the number by clicking the arrows, while also allowing adjustment by typing into a text entry.

A *DoubleSpinBox* can be used if the value to be stored is a double type.

22.1 Constructor

The SpinBox is constructed with the call:

```
spinbox = QSpinBox()
```

22.2 Methods

Setting a value on the SpinBox is done using:

```
spinbox.setValue(value)
```

If the *value* paramter is out of the minimum and maximum boundaries, the value will be adjusted so that it fits between the minimum and maximum.

Retrieval of the value set in the SpinBox is fetched via:

```
spinbox.value()
```

Minimum and maximum values are defined for the SpinBox using:

```
spinbox.setMinimum(value)
spinbox.setMaximum(value)
```

Alternatively, the range can be defined using a single call:

```
spinbox.setRange(minimum, maximum)
```

If required, the minimum and maximum values permissible in the SpinBox are found by calling:

```
spinbox.minimum()
spinbox.maximum()
```

A prefix and suffix can be displayed within the SpinBox:

```
spinbox.setPrefix(suffix)
spinbox.setSuffix(suffix)
```

The *prefix* and *suffix* parameters should be set to a string. It is useful when displaying a unit associated with the value (e.g. “mph”, “cm”).

By default, the adjustment arrows change the displayed value by 1. The step can be changed with:

```
spinbox.setSingleStep(value)
```

22.3 Example

Below is an example of a SpinBox:

Download: `SpinBox`

CHAPTER

TWENTYTHREE

DOUBLESPINBOX

A DoubleSpinBox is much like a regular *SpinBox*, however it is used to handle double type numbers. It supports numerical entry via the keyboard, or using the adjustment buttons built into the widget.

23.1 Constructor

The DoubleSpinBox widget is constructed with the call:

```
doublespinbox = QDoubleSpinBox()
```

23.2 Methods

Setting a value on the DoubleSpinBox is done using:

```
doublespinbox.setValue(value)
```

If the *value* paramter is out of the minimum and maximum boundaries, the value will be adjusted so that it fits between the minimum and maximum.

The value set in the DoubleSpinBox is retrievable via the use of:

```
doublespinbox.value()
```

Minimum and maximum values are defined for the DoubleSpinBox using:

```
doublespinbox.setMinimum(value)
doublespinbox.setMaximum(value)
```

If both minimum and maximum values are required, the range can be defined in a single method:

```
doublespinbox.setRange(minimum, maximum)
```

A prefix and suffix can be displayed within the DoubleSpinBox:

```
doublespinbox.setPrefix(suffix)
doublespinbox.setSuffix(suffix)
```

The *prefix* and *suffix* parameters should be set to a string. It is useful when displaying a unit associated with the value (e.g. “mph”, “cm”).

By default, the adjustment arrows change the displayed value by 1. The step can be changed with:

```
doublespinbox.setSingleStep(value)
```

23.3 Example

Below is an example of a DoubleSpinBox:

Download: `DoubleSpinBox`

CHAPTER

TWENTYFOUR

LCDNUMBER

LCDNumber is a display widget typically used for showing numbers with an LCD screen-like (e.g. calculator, watch) appearance.

24.1 Constructor

The LCDNumber is constructed via the call:

```
lcdnumber = QLCDNumber()
```

24.2 Methods

The contents to be displayed on the widget is set by:

```
lcdnumber.display(number)
lcdnumber.display(text)
```

The *number* argument can be set to an integer or float value. Alternatively, a *text* value can be displayed by passing a string.

Retrieval of the value from the LCDNumber is done using:

```
lcdnumber.value()
```

LCDNumber supports a number of modes including decimal, hex, oct, and binary which are set via:

```
lcdnumber.setMode(mode)
```

The *mode* should be set to one of:

- `Bin`
- `Oct`
- `Dec` (default)
- `Hex`

Also provides are convenice functions to enable each of the supported modes above:

```
lcdnumber.setBinMode()
lcdnumber.setOctMode()
lcdnumber.setDecMode()
lcdnumber.setHexMode()
```

The display of the decimal point can be configured with the method:

```
lcdnumber.setSmallDecimalPoint(small)
```

When *small* is set to `True`, the point is drawn between the two numbers. When `False`, the decimal point occupies a full digit position.

24.3 Example

Below is an example of a LCDNumber:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        lcdnumber = QLCDNumber()
        lcdnumber.display(4.5792)
        layout.addWidget(lcdnumber, 0, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `LCDNumber`

CHAPTER

TWENTYFIVE

IMAGE

The Image widget provides a way to displays images within a Qt application.

Note: There are actually four classes which handle the loading of images. These are:

- `QImage` - optimised for input/output.
- `QPixmap` - designed for showing images on screen.
- `QBitmap` - inherits from QPixmap with a depth of 1.
- `QPicture` - paint device to record and replay QPainter commands.

CHAPTER

TWENTYSIX

SPACERITEM

A SpacerItem provides a blank space in a layout. In most cases, the SpacerItem is not required as both the *BoxLayout* and *GridLayout* containers provide spacing declarations.

26.1 Constructor

The constructor for the SpacerItem is:

```
spaceritem = QSpacerItem()
```

26.2 Example

Below is an example of a SpacerItem:

Download: `SpacerItem`

CHAPTER

TWENTYSEVEN

PROGRESSBAR

A ProgressBar is used to show the completion state of a process. It is typically drawn using an empty box which fills as the job completes, coupled with a percentage value or textual description.

Use of a ProgressBar is recommended when a job may take some time, to ensure that the user is kept up-to-date on the state of the application.

27.1 Constructor

Construction of the ProgressBar is done with the call:

```
progressbar = QProgressBar()
```

27.2 Methods

The minimum and maximum values held by the ProgressBar are defined with:

```
progressbar.setMinimum(minimum)
progressbar.setMaximum(maximum)
```

The minimum and maximum values can also be retrieved:

```
progressbar.minimum()
progressbar.maximum()
```

The current value state of the ProgressBar is retrievable via:

```
progressbar.value()
```

Setting the value will typically be done by the application using:

```
progressbar.setValue(value)
```

The *value* parameter should be set to an integer value.

Orienting the ProgressBar is done with:

```
progressbar.setOrientation(orientation)
```

The *orientation* parameter should be set to one of:

- `Qt.Horizontal`
- `Qt.Vertical`

When the ProgressBar is horizontally oriented, the bar fills from left to right while the vertically oriented ProgressBar fills from top to bottom. This can be inverted via:

```
progressbar.setInvertedAppearance(appearance)
```

The completion percentage value can be set visible or not by using:

```
progressbar.setTextVisible(visible)
```

Changing the text displayed within the widget can be done with:

```
progressbar.setFormat(format)
```

The *format* value takes a string of text. The following modifiers are used to display the appropriate dynamic information:

- `%p` - percentage completion
- `%v` - current value
- `%m` - total number of steps

If required, the text can be retrieved by calling:

```
progressbar.format()
```

Reverting to the default text format of a percentage value can be done using:

```
progressbar.resetFormat()
```

27.3 Example

Below is an example of a ProgressBar:

Download: `ProgressBar`

CHAPTER

TWENTYEIGHT

PROGRESSDIALOG

The ProgressDialog is similar to the *ProgressBar*, with the ProgressBar portion of the widget placed in a dialog window. It is often used when the running process will require the user to wait, with the rest of the application being unavailable to use.

28.1 Constructor

Construction of the ProgressDialog is made using:

```
progressdialog = QProgressDialog()
```

28.2 Methods

Setting the value of the progress completion is made using the method:

```
progressdialog.setValue(value)
```

The value can also be retrieved with:

```
progressdialog.value()
```

Minimum and maximum values are also required to be assigned to the ProgressDialog to define the range of values permitted:

```
progressdialog.setMinimum(minimum)
progressdialog.setMaximum(maximum)
```

The ability to automatically close the ProgressDialog is made using:

```
progressdialog.setAutoClose(close)
```

A cancel button can be added to the ProgressDialog via:

```
progressdialog.setCancelButton(button)
```

The *button* argument should be set to an appropriate *PushButton*.

Checking whether a ProgressDialog was canceled by the user can be done using the call:

```
progressdialog.wasCanceled()
```

If `True` is returned, the user canceled the running process.

CHAPTER

TWENTYNINE

TOOLBAR

A Toolbar typically provides common shortcuts to features of an application (e.g. open file, find, zoom) and is usually displayed above the main content of the application.

29.1 Methods

Widgets are inserted into the Toolbar using:

```
toolbar.addWidget(widget)
toolbar.insertWidget(action, widget)
```

The *action* parameter within the `.insertWidget()` method should be set to an appropriate `action` object.

Separators allowing widgets to be grouped neatly are attached to the Toolbar with:

```
toolbar.addSeparator()
toolbar.insertSeparator(action)
```

All items within the Toolbar can be cleared using:

```
toolbar.clear()
```

By default, Toolbar widgets are usually horizontally orientated. The orientation can be set with:

```
toolbar.setOrientation(orientation)
```

The *orientation* value should be set vertically or horizontally with one of the following:

- `Qt.Vertical`
- `Qt.Horizontal`

Newly-created Toolbar widgets have a handle on the left which provides for detaching the toolbar and allowing the user to position it elsewhere. This can be disabled via:

```
toolbar.setMovable(movable)
```

When the *movable* is set to `False`, the grab handle is hidden and the Toolbar is not able to be moved.

In some cases, it may be useful to allow the Toolbar to be floated in its own window:

```
toolbar.setFloatable(floatable)
```

The widget associated with an Action object can be found using:

```
toolbar.widgetForAction(action)
```

29.2 Example

Below is an example of a Toolbar:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        toolbar = QToolBar()
        layout.addWidget(toolbar)

        toolbutton = QToolButton()
        toolbutton.setText("Button 1")
        toolbutton.setCheckable(True)
        toolbutton.setAutoExclusive(True)
        toolbar.addWidget(toolbutton)

        toolbutton = QToolButton()
        toolbutton.setText("Button 2")
        toolbutton.setCheckable(True)
        toolbutton.setAutoExclusive(True)
        toolbar.addWidget(toolbutton)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Toolbar`

CHAPTER

THIRTY

TOOLBOX

The ToolBox widget is a container which displays groups of items separated by tabs, with the item consisting of the text identifying the item, and an optional icon. The ToolBox is commonly used in applications where there are too many items to place in a *Toolbar*.

30.1 Constructor

The ComboBox widget is created by defining:

```
toolbox = QToolBox()
```

30.2 Methods

Items can be added to the ToolBox via two methods:

```
toolbox.addItem(child, label)
toolbox.addItem(child, icon, label)
toolbox.insertItem(index, child, label)
toolbox.insertItem(index, child, icon, label)
```

The *child* parameter is the widget to be added to the ToolBox. The *label* value is the item name to be displayed on the ToolBox. An *icon* can also be added to each item using the *Icon* object. The `.insertItem()` method also takes an *index* parameter which indicates the position at which the child should be added.

Items can also be removed:

```
toolbox.removeItem(index)
```

The *index* value indicates the position of the child widget to be removed, with `0` indicating the first item.

It may be useful to get the active item index or widget with the methods:

```
toolbox.currentIndex()
toolbox.currentWidget()
```

The number of items contained in the ToolBox can be fetched using:

```
toolbox.count()
```

To disable (grey-out) an item and prevent it being accessed, call:

```
toolbox.setItemEnabled(index, state)
```

The *index* value indicates which item is to be disabled and the state, when set to `False` will disable the item.

Item attributes can also be changed after add/insert with:

```
toolbox.setText(index, label)
toolbox.setIcon(index, icon)
```

A tooltip, which is displayed when the user hovers over a child, can be associated with each item:

```
toolbox.setToolTip(index, text)
```

The *index* value indicates the child which is to receive the tooltip. The *text* value is the string of text to be attached.

The text, icon and tooltip can also be retrieved from the ToolBox by calling:

```
toolbox.itemText(index)
toolbox.itemIcon(index)
toolbox.itemToolTip(index)
```

The *index* value should be set to the number of the item which is to be retrieved.

To find the index number for a given child widget call:

```
toolbox.indexOf(widget)
```

Alternatively, the widget for a given index number is found using:

```
toolbox.widget(index)
```

30.3 Example

Below is an example of a ToolBox:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        toolbox = QToolBox()
        layout.addWidget(toolbox, 0, 0)

        label = QLabel()
        toolbox.addItem(label, "Honda")
        label = QLabel()
        toolbox.addItem(label, "Toyota")
        label = QLabel()
        toolbox.addItem(label, "Mercedes")

app = QApplication(sys.argv)

screen = Window()
screen.show()
```

```
sys.exit(app.exec_())
```

Download: `ToolBox`

CHAPTER

THIRTYONE

TOOLBUTTON

The ToolButton widget provides a button which can be added to a *Toolbar* or *ToolBox* container. They are used commonly for quick access to common functions such as saving a document, or finding a string of text.

31.1 Constructor

Construction of the ToolButton is made using:

```
toolbutton = QToolButton()
```

31.2 Methods

Text can be added to the ToolButton by calling:

```
toolbutton.setText(text)
```

An icon can also be added to the ToolButton with:

```
toolbutton.setIcon(icon)
```

The *icon* parameter should be set to an appropriate *Icon* object.

ToolButton widgets can also be made checkable. This allows them to be in either a pressed or unpressed state, and is useful for indicating a true or false state. The function can be set using:

```
toolbutton.setCheckable(checkable)
```

When *checkable* is set to `True`, the ToolButton will appear depressed when clicked.

The checked state of the ToolButton can then be retrieved by calling:

```
toolbutton.isChecked(checked)
```

Progamatically, the ToolButton when made checkable can be pressed with:

```
toolbutton.setDown(down)
```

A *Menu* object can be added to the ToolButton to provide a dropdown menu:

```
toolbutton.setMenu(menu)
```

If a menu is in use with the ToolButton, the way the menu pops up can be configured by:

```
toolbutton.setPopupMode(mode)
```

The *mode* value should be set to one of:

- `QToolButton.DelayPopup` - the Menu is shown when the ToolButton is pressed and held for a set time.
- `QToolButton.MenuButtonPopup` - show an arrow next to the ToolButton, which displays the Menu object when clicked.
- `QToolButton.InstantPopup` - display the Menu immediately when the ToolButton item is clicked.

31.3 Example

An example of the ToolButton in use can be found in the ToolBar example.

CHAPTER

THIRTYTWO

MENUBAR

A MenuBar provides a horizontal bar which is used as a container for other widgets. Typically these will be Button and Menu combinations which provide additional options for the application functionality.

32.1 Constructor

The MenuBar can be constructed using:

```
menubar = QMenuBar()
```

32.2 Methods

Actions, which perform an associated function when clicked can be added to the MenuBar with a simple text label:

```
action = menubar.addAction(label)
```

When the Action is defined, it is also returned allowing the use of methods defined in the `action` documentation.

Alternatively, if a menu should be displayed on click with many options, the following can be called:

```
menu = menubar.addMenu(label)
```

As with the Action example, adding a menu returns a *Menu* item.

Items are also removable by:

```
menubar.removeAction(action)
```

The *action* value should be set to the name of the Action to be removed.

Separators are supported by the MenuBar with:

```
menubar.addSeparator()
```

All the actions specified for the MenuBar may be cleared via:

```
menubar.clear()
```

32.3 Example

Below is an example of an MenuBar. This also contains examples of the Action and Menu widgets as they are closely associated with the MenuBar.

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

from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        menubar = QMenuBar()
        layout.addWidget(menubar, 0, 0)

        actionFile = menubar.addMenu("File")
        actionFile.addAction("New")
        actionFile.addSeparator()
        actionFile.addAction("Quit")

        menubar.addMenu("Edit")
        menubar.addMenu("View")
        menubar.addMenu("Help")

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `MenuBar`

CHAPTER

THIRTYTHREE

MENU

The Menu item provides the base layer for the menu items which are displayed on it. This can include single-click items, check and radio items, or additional menus.

33.1 Constructor

The Menu is constructed with the call:

```
menu = QMenu()
```

Note: When building a menubar for use in an application, the Menu item would not need to be manually constructed as it can be obtained from an existing menu action item.

33.2 Methods

Adding an item to the Menu with a simple text entry is done with:

```
action = menu.addAction(text)
```

The *text* value should be set to the purpose of the action item. When called, it also returns the object for the item, allowing other `action` methods to be applied.

Another menu can be added to the Menu with:

```
menu = menu.addMenu()
menu = menu.insertMenu(action)
```

The `.insertMenu()` method takes an action parameter which determines the item on which the new menu should be inserted before.

The Menu can also contain sections which are useful for grouping items:

```
menu.addSection(text)
menu.insertSection(action, text)
```

The *text* parameter is set for the title of the section. If using the `.insertSection()`, the *action* argument is also needed which indicates another item where the section should be inserted before.

To add a separator between items in the Menu use:

```
menu.addSeparator()
```

All the items contained by the Menu can be cleared with the method:

```
menu.clear()
```

The tearoff functionality allows menus to be floated in a window for easy access. This can be enabled on a menu with:

```
menu.setTearOffEnabled(enabled)
```

A title should also be set when using the tearoff functionality, to ensure the floating window has an appropriate title:

```
menu.setTitle(title)
```

CHAPTER

THIRTYFOUR

TABWIDGET

The TabWidget provides a container with multiple pages which are switchable via tabs. Each page can contain a single widget or other containers. The TabWidget is commonly found in multi-document applications such as web browsers or word processors.

34.1 Constructor

The TabWidget is constructed using:

```
tabwidget = QTabWidget()
```

34.2 Methods

Tabs are added to the TabWidget via several methods:

```
tabwidget.addTab(child, label)
tabwidget.insertTab(index, child, label)
```

The `.addTab()` method adds each tab in the order the code is executed whereas the `.insertTab()` method allows an *index* value indicating the location to insert the tab, with the first position identified as `0`. The *child* parameter is the name of the child object to be added to the tab. Finally, the *label* parameter is the text to be displayed on the tab itself.

Additionally, an icon can be added to each tab with:

```
tabWidget.addTab(child, icon, label)
tabWidget.insertTab(index, child, icon, label)
```

The *index*, *child*, and *label* arguments remain as above. The *icon* parameter should be set to an appropriate *Icon* object.

Tabs are removed from the TabWidget via:

```
tabWidget.removeTab(index)
```

The *index* value is the position of the tab within the TabWidget.

All the tabs currently held by the TabWidget can be removed with:

```
tabWidget.clear()
```

The text displayed on each tab can be configured post-add with:

```
tabWidget.setTabText(index, text)
```

The number of tabs contained can be counted using:

```
tabWidget.count()
```

If the TabWidget contains less than two tabs, the tab bar can be configured to hide:

```
tabWidget.tabBarAutoHide(autohide)
```

When *autohide* is set to `True`, the tab bar will be hidden when there are fewer than two tabs being displayed.

In some cases, individual tabs should be removable. A close button can be added to each tab using:

```
tabWidget.setTabsClosable(closable)
```

Each tab can be assigned a ToolTip and/or *WhatsThis* to indicate the purpose of the tab with:

```
tabWidget.setTabToolTip(index, text)
tabWidget.setWhatsThis(index, text)
```

The *index* argument specifies which tab should receive the *text* parameter, with `0` specifying the first tab held by the TabWidget.

In some cases, individual tabs will need to be made inaccessible to the user. This is done by "greying-out" via:

```
tabWidget.setTabEnabled(index, enabled)
```

When *enabled* is set to `True`, the user will not be able to interact with it, though the content may still be available to view.

A check can be run on whether a tab is enabled with:

```
tabWidget.isTabEnabled(index)
```

Tabs may also be required to be movable. This can be set via:

```
tabWidget.setMovable(movable)
```

By default, tabs are not movable by the user, but when *movable* is set to `True`, the user can drag-and-drop each tab into a new place.

The default appearance is to display all tabs at the top of the widget. This can be configured with:

```
tabWidget.setTabPosition(position)
```

The *position* value should be set to one of:

- `QTabWidget.North` - draw tabs at top (default).
- `QTabWidget.South` - draw tabs at bottom.
- `QTabWidget.East` - draw tabs on left.
- `QTabWidget.West` - draw tabs on right.

34.3 Example

Below is an example of a TabWidget:

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

from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
```

```
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        label1 = QLabel("Example content contained in a tab.")
        label2 = QLabel("More example text in the second tab.")

        tabwidget = QTabWidget()
        tabwidget.addTab(label1, "Tab 1")
        tabwidget.addTab(label2, "Tab 2")
        layout.addWidget(tabwidget, 0, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `TabWidget`

CHAPTER

THIRTYFIVE

TABBAR

A TabBar provides the drawing of tabs, with common usage in tabbed dialogs. It is similar to the *TabWidget* which is a ready-made solution, whereas the TabBar provides more configuration for layout and style.

35.1 Constructor

The construction method for the TabBar is:

```
tabbar = QTabBar()
```

35.2 Methods

A tab is added to the TabBar using one of four methods:

```
tabbar.addTab(text)
tabbar.addTab(icon, text)
tabbar.insertTab(index, text)
tabbar.insertTab(index, icon, text)
```

The *text* parameters passes the string to be displayed on the newly created tab. The *icon* argument specifies an *Icon* object to be displayed alongside the text. The `.insertTab()` methods also take an *index* parameter which specifies where in the TabBar the new tab will be inserted.

Removal of tabs is done by the call:

```
tabbar.removeTab(index)
```

The *index* value specifies the current position of the tab to be removed, with `0` used for the first tab.

If the user should be able to move tabs within the TabBar, call:

```
tabbar.setMovable(movable)
```

A tab can be programatically moved via:

```
tabbar.moveTab(from, to)
```

The *from* and *to* values indicate the position of the tab, from its current position to the new one.

In some circumstances such as a preferences dialog, some tabs may be made unavailable. This is done by:

```
tabbar.setTabEnabled(index, enabled)
```

The *index* passes the position of the tab be made enabled or disabled. When *enabled* is set to `False`, the tab will be greyed-out and inaccessible.

The icon and text on a tab can be defined after it has been added:

```
tabbar.setTabText(index, text)
tabbar.setTabIcon(index, icon)
```

Some applications such as web browsers require that tabs be closed. This is configurable via:

```
tabbar.setTabsClosable(closable)
```

Then *closable* is set to `True`, a close button is added to the tab.

A *ToolTip* and *WhatsThis* can be added with the methods:

```
tabbar.setTabToolTip(index, tooltip)
tabbar.setTabWhatsThis(index, whatsthis)
```

The *tooltip* and *whatsthis* parameters should be set to a string.

By default, the TabBar should always be visible. In may be preferential for the TabBar to be hidden when less than two tabs are visible:

```
tabbar.setAutoHide(autohide)
```

Retrieval of the number of tabs currently held by the TabBar with:

```
tabbar.count()
```

The active tab can be fetched and retrieved using the calls:

```
tabbar.currentIndex()
tabbar.setCurrentIndex(index)
```

35.3 Example

Below is an example of an TabBar:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        tabbar = QTabBar()
        tabbar.addTab("Tab 1")
        tabbar.addTab("Tab 2")
        tabbar.addTab("Tab 3")
        layout.addWidget(tabbar, 0, 0)

app = QApplication(sys.argv)
```

```
screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: TabBar

CHAPTER

THIRTYSIX

STACKEDWIDGET

The StackedWidget is a container which displays a single page at a time. A left-hand panel provides access to the pages which are then displayed to the right.

36.1 Constructor

Construction of the StackedWidget is done using:

```
stackedwidget = QStackedWidget()
```

36.2 Methods

To add an item to the StackedWidget use:

```
stackedwidget.addWidget(widget)
stackedwidget.insertWidget(index, widget)
```

The *index* value should be set to the numerical position identifying where the widget should be inserted.

Removal of the widget from the StackedWidget is done using:

```
stackedwidget.removeWidget(widget)
```

The index value or the widget currently visible widget within the StackedWidget is obtained via either:

```
stackedwidget.currentIndex()
stackedwidget.currentWidget()
```

The current page visible can be set by specifying the page index or widget:

```
stackwidget.setCurrentIndex(index)
stackwidget.setCurrentWidget(widget)
```

To retrieve the number of widgets held by the StackedWidget call:

```
stackedwidget.count()
```

36.3 Example

Below is an example of a StackedWidget:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.stackedwidget = QStackedWidget()
        layout.addWidget(self.stackedwidget, 0, 0)

        for x in range(1, 4):
            label = QLabel("Stack Child %i" % (x))
            self.stackedwidget.addWidget(label)

            button = QPushButton("Stack %i" % (x))
            button.page = x
            button.clicked.connect(self.on_button_clicked)
            layout.addWidget(button, x, 0)

    def on_button_clicked(self):
        button = self.sender()
        self.stackedwidget.setCurrentIndex(button.page - 1)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `StackedWidget`

CHAPTER

THIRTYSEVEN

DOCKWIDGET

A DockWidget provides a widget which is able to be docked inside the main window, or placed in its own separate window. The widget is useful for holding widgets where it would be useful to separate them from the main interface.

37.1 Constructor

The widget is constructed via:

```
dockwidget = QDockWidget()
```

37.2 Methods

Adding the child widget is done using:

```
dockwidget.setWidget(widget)
```

The child widget attached can be retrieved if required:

```
dockwidget.widget()
```

The palette can be set to floating programmatically via:

```
dockwidget.setFloating(float)
```

The floating status of the DockWidget can also be retrieved with:

```
dockwidget.isFloating()
```

A number of customisations can be made to the DockWidget with the method:

```
dockwidget.setFeatures(features)
```

The features list can include the following:

- `QDockWidget.DockWidgetClosable` - allow the DockWidget to be closed.
- `QDockWidget.DockWidgetMovable` - allow the DockWidget to be moved.
- `QDockWidget.DockWidgetFloatable` - allow the DockWidget to be floated.
- `QDockWidget.DockWidgetVerticalTitleBar` - set the title bar vertically.
- `QDockWidget.DockWidgetNoDockWidgetFeatures` - turn off all features.

An arbitary widget can be set for use in the DockWidget title bar. This could be a container containing several widgets, or a single widget. They are set via:

```
dockwidget.setTitleBarWidget(widget)
```

The title displayed on the floating window is defined by using:

```
dockwidget.setWindowTitle(title)
```

When the DockWidget is docked, the title is displayed vertically alongside the frame of the widget.

If a widget is to be added as the title rather than a simple label, the add method is:

```
dockwidget.setTitleBarWidget(widget)
```

37.3 Example

Below is an example of a DockWidget:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        dockwidget = QDockWidget()
        dockwidget.setFeatures(QDockWidget.DockWidgetClosable | QDockWidget.DockWidgetVerticalTitleBa
        layout.addWidget(dockwidget)

        treewidget = QTreeWidget()
        dockwidget.setWidget(treewidget)

        label = QLabel("DockWidget is docked")
        layout.addWidget(label)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: DockWidget

CHAPTER

THIRTYEIGHT

FORMLAYOUT

The FormLayout provides a class layout to handle input widgets and their associated labels. The children are laid out in two columns, with the label column handling the label and the right column providing space for the input widgets such as text entries or spin boxes.

38.1 Constructor

The construction call for the FormLayout is:

```
formlayout = QFormLayout()
```

38.2 Methods

Rows can be added to the container using:

```
formlayout.addRow(label, widget)
formlayout.addRow(text, widget)
formlayout.addRow(widget)
```

An alternative method of adding rows allows for the position of the new row being inserted to be defined:

```
formlayout.insertRow(row, label, widget)
formlayout.insertRow(row, text, widget)
formlayout.insertRow(row, widget)
```

The *widget* parameter can be a widget or another container. The *label* should be set to the *Label* which is to be shown. Alternatively, *text* can be defined which automatically creates the label.

The spacing provided vertically or horizontally, or both can be set via:

```
formlayout.setVerticalSpacing(spacing)
formlayout.setHorizontalSpacing(spacing)
formlayout.setSpacing(spacing)
```

The handling of how the fields grow based on size is controlled via the method:

```
formlayout.setFieldGrowthPolicy(policy)
```

The *policy* parameter can be set to one of the following:

- `QFormLayout.FieldsStayAtSizeHint` - the fields never grow beyond their size hint.

- `QFormLayout.ExpandingFieldsGrow` - when set to expand, the fields will grow to fill the available space.
- `QFormLayout.AllNonFixedFieldsGrow` - all fields will grow to fill the available space.

38.3 Example

Below is an example of a FormLayout:

Download: `FormLayout`

CHAPTER

THIRTYNINE

COMBOBOX

A ComboBox provides a dropdown menu attached to a button, providing a list of options of which one can be selected by the user.

39.1 Constructor

The ComboBox widget is created by defining:

```
combobox = QComboBox()
```

39.2 Methods

Individual items are added to the ComboBox using the methods:

```
combobox.addItem(text)
combobox.insertItem(index, text)
```

The *text* value should be set to the string of text which is to be added to the ComboBox. The `.insertItem()` method also allows for an *index* value to be specified which indicates where the item will be inserted.

An alternative is to add multiple items with a single method:

```
combobox.addItems(text, text, text...)
combobox.insertItems(index, text, text, text...)
```

Separators can be inserted into a specific position within the ComboBox popup with:

```
combobox.insertSeparator(index)
```

Removal of items is done with the method:

```
combobox.removeItem(index)
```

The *index* defines the location of the item to be removed held within the ComboBox, with `0` pointing to the first item.

The currently selected index or text is retrievable with the following:

```
combobox.currentIndex()
combobox.currentText()
```

To retrieve the number of items held within the ComboBox use:

```
combobox.count()
```

The number of items permitted, and the maximum visible within the ComboBox is set by:

```
combobox.setMaxCount(maximum)
combobox.setMaxVisibleItems(maximum)
```

The *maximum* value should be an integer value indicating the limit. If the number of items added is greater than the maximum amount, the extra items are truncated.

The ComboBox popup menu can be shown or hidden programmatically with:

```
combobox.showPopup()
combobox.hidePopup()
```

Auto-completion functionality with the *Completer* object can be added to the ComboBox widget with:

```
combobox.setCompleter(completer)
```

By default, duplicate items are not allowed in the ComboBox, however this can be toggled using:

```
combobox.setDuplicatesEnabled(enable)
```

The ComboBox can display an integrated *LineEdit* to allow the user to enter items which are not provided in the dropdown menu using:

```
combobox.setEditable(editable)
```

A LineEdit manually constructed can be added ot the ComboBox via the method:

```
combobox.setLineEdit(lineedit)
```

The LineEdit object can be obtained from the ComboBox by calling:

```
combobox.lineEdit()
```

Control over whether a user-added item should appear in the ComboBox can be set with the method:

```
combobox.setInsertPolicy(policy)
```

The *policy* parameter should be set to one of the following:

- `QComboBox.NoInsert` - the item will not be inserted into the ComboBox.
- `QComboBox.InsertAtTop` - item will be added as the first in the ComboBox.
- `QComboBox.InsertAtCurrent` - item will be replaced by the new string.
- `QComboBox.InsertAtBottom` - item will be added as the last in the ComboBox.
- `QComboBox.InsertAfterCurrent` insert after current item in the ComboBox.
- `QComboBox.InsertBeforeCurrent` - insert before the current item in the ComboBox.
- `QComboBox.InsertAlphabetically` - insert item in alphabetical ordering.

If the data is being held by a model, this can be attached to the ComboBox with:

```
combobox.setModel(model)
```

The column number of the data within the model should also be specified:

```
combobox.setModelColumn(column)
```

By default, the *column* value is automatically set to `0` to indicate the first column of data. If the data to be displayed is held in a different column, define the integer value for that column.

39.3 Example

Below is an example of an ComboBox:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.combobox = QComboBox()
        self.combobox.addItem("Birch")
        self.combobox.addItem("Oak")
        self.combobox.addItem("Sycamore")
        self.combobox.currentTextChanged.connect(self.combobox_changed)
        layout.addWidget(self.combobox)

    def combobox_changed(self):
        text = self.combobox.currentText()
        print(text)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `ComboBox`

CHAPTER

FORTY

COMPLETER

The Completer object is used to provide auto-completions when text is entered into some widgets such as the *LineEdit* or *ComboBox*. When a user begins to type, the model content is matched and suggestions are provided.

40.1 Constructor

A Completer is constructed using:

```
completer = QCompleter()
```

The data model can be added post-construction, however it can be defined at construction time by using:

```
completer = QCompleter(model)
```

40.2 Methods

Data used by the Completer is held in a model, which is attached by calling:

```
completer.setModel(model)
```

The model attached to the Completer can also be retrieved with:

```
completer.model()
```

In some cases, the data model may contain multiple columns. By default, the completer uses the first column (0), however this can be changed by the method:

```
completer.setCompletionColumn(column)
```

The completion method set on the Completer is set using:

```
completer.setCompletionMode(mode)
```

The *mode* defined should be set to one of:

- `QCompleter.PopupCompletion` - completions are displayed in a dropdown menu.
- `QCompleter.InlineCompletion` - completions appear inline as selected text.
- `QCompleter.UnfilterPopupCompletion` - completions are displayed in a dropdown menu with the most likely suggestion indicated as current.

By default, seven items are displayed in the completion. An alternative value can be set using:

```
completer.setMaxVisibleItems(maximum)
```

In some cases, it may be preferable to control whether the completion is sensitive or insensitive via:

```
completer.setCaseSensitivity(sensitivity)
```

The *sensitivity* constant should be defined as one or:

- `Qt.CaseInsensitive`
- `Qt.CaseSensitive`

40.3 Example

Below is an example of a Completer:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        names = ["George", "Marcus", "Samantha", "Steven", "Maria"]

        completer = QCompleter(names)

        self.lineedit = QLineEdit()
        self.lineedit.setCompleter(completer)
        layout.addWidget(self.lineedit, 0, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Completer`

CHAPTER

FORTYONE

CALENDAR

The Calender widget provides a way to select a date and show a date to the user.

41.1 Constructor

The Calendar is constructed using the call:

```
calendar = QCalendarWidget()
```

41.2 Methods

A number of functions are available for changing the date relative to the current date with:

```
calendar.showToday()
calendar.showSelectedDate()
calendar.showNextMonth()
calendar.showNextYear()
calendar.showPreviousMonth()
calendar.showPreviousYear()
```

The selected date can be retrieved from the Calendar with:

```
calendar.selectedDate()
```

This returns a *Date* object which contains a number of associated methods for retrieving the date.

The current page, determined by the specified month and year can be set via:

```
calendar.setCurrentPage(month, year)
```

The minimum and maximum dates viewable within the Calendar can be set with:

```
calendar.minimumDate()
calendar.maximumDate()
```

A Date object is returned for both methods which contains the minumum and maximum date ranges.

Minimum and maximum dates can also be defined via the Date object with the methods:

```
calendar.setMinimumDate(date)
calendar.setMaximumDate(date)
```

By default, the Calendar allows the date to be changed. It is possible to prevent the Calendar from being changed using:

```
calendar.setDateEditEnabled(enabled)
```

When *enabled* is set to `False`, the user is no longer able to modify the Calendar, however it can still be used to display dates set programatically.

The view of the Calendar can be customised by showing or hiding both the grid lines and navigation bar:

```
calendar.isGridVisible(visible)
calendar.isNavigationBarVisible(visible)
```

41.3 Signals

When the date selection is changed, either by the user changing the date or by programmatically changing the date, the `.selectionChanged()` signal is emitted.

41.4 Example

Below is an example of a Calendar:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        calendar = QCalendarWidget()
        layout.addWidget(calendar)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Calendar`

CHAPTER

FORTYTWO

DATEEDIT

The DateEdit widget allows date information to be displayed and changed.

42.1 Constructor

Construction of the DateEdit is made using:

```
dateedit = QDateEdit()
```

42.2 Methods

Date information can be set onto the widget using the method:

```
dateedit.setDate(date)
```

The current date set on the DateEdit widget is fetched using:

```
dateedit.date()
```

A range of permissible dates is defined on the DateEdit by calling:

```
dateedit.setMinimumDate(date)
dateedit.setMaximumdate(date)
```

In both methods, the *date* parameter should be an appropriate *Date* object which defines the date to set.

If required, the minimum and maximum range can be cleared individually using:

```
dateedit.clearMinimumDate()
dateedit.clearMaximumDate()
```

42.3 Example

Below is an example of a DateEdit:

Download: `DateEdit`

CHAPTER

FORTYTHREE

TIMEEDIT

The TimeEdit widget provides an editable box from which time value can be displayed and edited.

43.1 Constructor

The constructor for the TimeEdit is:

```
timeedit = QTimeEdit()
```

43.2 Methods

The time is settable on the widget via:

```
timeedit.setTime(time)
```

Time is also retrievable from the TimeEdit using:

```
timeedit.time()
```

Minimum and maximum permissable time values can be set to define a range with the methods:

```
timeedit.setMinimumTime(minimum)
timeedit.setMaximumTime(maximum)
```

The *minimum* and *maximum* values should be set to an appropriate *Time* object which contains the defined time values.

The ranges defined for minimum and maximum times are cleared with:

```
timeedit.clearMinimumTime()
timeedit.clearMaximumTime()
```

43.3 Example

Below is an example of a TimeEdit:

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

from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
import sys
```

```
class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        time = QTime()
        time.setHMS(13, 15, 40)

        timeedit = QTimeEdit()
        timeedit.setTime(time)
        layout.addWidget(timeedit, 0, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `TimeEdit`

CHAPTER

FORTYFOUR

DATETIMEEDIT

The DateTimeEdit widget provides the functionality of both the *DateEdit* and *TimeEdit* widgets in one, allowing both time and date information to be modified and displayed to the user.

44.1 Constructor

The construction of the DateTimeEdit is made via:

```
datetimeedit = QDateTimeEdit()
```

44.2 Methods

The currently displayed Date, Time, and DateTime objects can be obtained from the DateTimeEdit by calling the methods:

```
datetimeedit.date()
datetimeedit.dateTime()
datetimeedit.time()
```

Minimum and maximum dates and times can be defined which permit only a range to be accessed by:

```
datetimeedit.setMinimumDate(date)
datetimeedit.setMinimumDateTime(datetime)
datetimeedit.setMinimumTime(time)
datetimeedit.setMaximumDate(date)
datetimeedit.setMaximumDateTime(datetime)
datetimeedit.setMaximumTime(time)
```

The *date*, *time* and *datetime* parameters should be set to an appropriate object of the respective type *Date*, *Time*, and *DateTime*.

Date, Time and DateTime ranges can be defined via a single method using:

```
datetimeedit.setTimeRange(minimum, maximum)
datetimeedit.setDateTimeRange(minimum, maximum)
datetimeedit.setDateRange(minimum, maximum)
```

The minimum and maximum dates and times are retrieved via:

```
datetimeedit.minimumDate()
datetimeedit.minimumDateTime()
datetimeedit.minimumTime()
```

```
datetimeedit.maximumDate()
datetimeedit.maximumDateTime()
datetimeedit.maximumTime()
```

If required, the minimum and maximum defined objects from above can be cleared:

```
datetimeedit.clearMinimumDate()
datetimeedit.clearMinimumDateTime()
datetimeedit.clearMinimumTime()
datetimeedit.clearMaximumDate()
datetimeedit.clearMaximumDateTime()
datetimeedit.clearMaximumTime()
```

A *Calendar* widget can be added to the DateTimeEdit using:

```
datetimeedit.setCalendarWidget(widget)
```

CHAPTER

FORTYFIVE

DIALOG

A Dialog is effectively similar to a *Window* in appearance. It is commonly used in very simple applications, or as a sub-window (e.g. preferences) of an application.

45.1 Constructor

The Dialog window is constructed with the call:

```
dialog = QDialog()
```

45.2 Methods

In certain circumstances, it may be useful to prevent the user from interacting with any other windows apart from the dialog. This is called modal operation, and defaults to False. To set the Dialog as modal use:

```
dialog.setModal(modal)
```

The modal state of the Dialog can be retrieved using:

```
modal = dialog.isModal()
```

To allow users to easily resize the Dialog, a *SizeGrip* can be added:

```
dialog.sizeGripEnabled(enabled)
```

When *enabled* is set as `True`, the SizeGrip will be added.

45.3 Example

Below is an example of a Dialog:

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

from PyQt5.QtWidgets import *
import sys

class Dialog(QWidget):
    def __init__(self):
        QWidget.__init__(self)
```

```
        layout = QGridLayout()
        self.setLayout(layout)

        label = QLabel("This is a dialog.")
        layout.addWidget(label, 0, 0)

        buttonbox = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel)
        layout.addWidget(buttonbox)

app = QApplication(sys.argv)

screen = Dialog()
screen.show()

sys.exit(app.exec_())
```

Download: `Dialog`

CHAPTER

FORTYSIX

FILEDIALOG

The FileDialog widget provides a dialog useful for selecting of files. These are commonly used when a user wants to open or save a file within the application.

46.1 Constructor

Construction of the FileDialog widget is made using:

```
filedialog = QFileDialog()
```

46.2 Methods

The FileDialog can be opened using the method:

```
filedialog.open()
```

To define whether the dialog is to be used for opening or saving files call:

```
filedialog.setAcceptMode(mode)
```

The *mode* value should be set to one of:

- `QFileDialog.AcceptOpen`
- `QFileDialog.AcceptSave`

Text can be displayed in the FileDialog indicating the purpose with:

```
filedialog.setLabelText(text)
```

A default suffix can be added for display if no other suffix is currently in use via:

```
filedialog.setDefaultSuffix(suffix)
```

The *suffix* parameter should be a string and is commonly used to identify the type of file such as '.txt', '.odt', or '.png' for example.

Configuration of the displayed information within the FileDialog can be done with:

```
filedialog.setViewMode(mode)
```

The *mode* in this case can be set to:

- `QFileDialog.Detail` - display an icon, name, and details for each item.

- `QFileDialog.List` - display icon and name only.

In some cases, the requirement will be that the dialog only display certain file types. The *Dir* object can be set with:

```
filedialog.setFilter(filter)
```

46.3 Example

Below is an example of a FileDialog:

Download: `FileDialog`

CHAPTER

FORTYSEVEN

FONTDIALOG

The FontDialog widget provides a widget for selecting a font, including the font type, the size, and features such as bold or italic styling. The dialog provides an area for previewing the selected font.

47.1 Constructor

Construction of the FontDialog is made using:

```
fontdialog = QFontDialog(parent)
```

The *parent* argument supplied indicates the widget (i.e. window) which owns the FontDialog.

47.2 Methods

The FontDialog is opened using:

```
fontdialog.open()
```

The current font can be set onto the FontDialog with:

```
fontdialog.setCurrentFont(font)
```

Use of the *font* parameter requires a `font` object.

Retrieval of the font from the dialog is done via:

```
fontdialog.currentFont()
```

Alternatively, the returned font from the dialog when the user presses the OK button is able to be fetched using:

```
fontdialog.selectedFont()
```

Options customising the dialog state is done using:

```
fontdialog.setOptions(options)
```

The *options* value can be set to one or more of the following constants:

- `QFontDialog.NoButtons` - do not show any OK or Cancel buttons.
- `QFontDialog.DontUseNativeDialog` - use Qt dialog rather than the native platform dialog.
- `QFontDialog.ScalableFonts` - show scalable fonts.
- `QFontDialog.NonScalableFonts` - show non-scalable fonts.

- `QFontDialog.MonospacedFonts` - show monospaced fonts.
- `QFontDialog.ProportionalFonts` - show proportional fonts.

Retrieval of the options from the dialog is done by calling:

```
fontdialog.options()
```

47.3 Example

Below is an example of a FontDialog:

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

from PyQt5.QtWidgets import *
import sys

class FontDialog(QFontDialog):
    def __init__(self):
        QFontDialog.__init__(self)
        self.fontSelected.connect(self.on_font_selected)

    def on_font_selected(self):
        font = self.currentFont()

        print("Name:      %s" % (font.family()))
        print("Size:      %i" % (font.pointSize()))
        print("Italic:    %s" % (font.italic()))
        print("Underline: %s" % (font.underline()))
        print("Strikeout: %s" % (font.strikeOut()))

    def run(self):
        self.show()

app = QApplication(sys.argv)

screen = FontDialog()
screen.run()

sys.exit(app.exec_())
```

Download: `FontDialog`

CHAPTER

FORTYEIGHT

FONTCOMBOBOX

The FontComboBox widget provides a way for a user to select a font family from a dropdown list. It is often used within *Toolbar* widgets in applications such as word processors to allow the user to change fonts.

48.1 Constructor

The constructor for the FontComboBox is:

```
fontcombobox = QFontComboBox()
```

48.2 Methods

The font set on the FontComboBox is retrievable with:

```
fontcombobox.currentFont()
```

A font can also be preset programmatically using:

```
fontcombobox.setCurrentFont(font)
```

The *font* parameter should be set to a `font` object holding the related information.

By default, all fonts installed on the system are shown. These can be filtered using:

```
fontcombobox.setFontFilters(filters)
```

The *filters* parameter should be set to one of:

- `QFontComboBox.AllFonts` - show all fonts.
- `QFontComboBox.ScalableFonts` - show scalable fonts.
- `QFontComboBox.NonScalableFonts` - show non-scalable fonts.
- `QFontComboBox.MonospacedFonts` - show monospaced fonts.
- `QFontComboBox.ProportionalFonts` - show proportional fonts.

48.3 Example

Below is an example of a FontComboBox:

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

from PyQt5.QtWidgets import *
import sys

class Dialog(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        fontcombobox = QFontComboBox()
        fontcombobox.currentFontChanged.connect(self.on_font_changed)
        layout.addWidget(fontcombobox)

    def on_font_changed(self):
        fontcombobox = self.sender()
        font = fontcombobox.currentFont()

        print("Selected font: %s" % (font.family()))

app = QApplication(sys.argv)

screen = Dialog()
screen.show()

sys.exit(app.exec_())
```

Download: `FontComboBox`

CHAPTER

FORTYNINE

COLORDIALOG

The ColorDialog widget provides a colour chooser positioned within a dialog. This allows the user to select a range of colours from a palette or by entering colour values.

Note: By default, the ColorDialog used is Qt's own native widget. It is also possible to use the platform native dialog instead, however this may behave differently.

49.1 Constructor

To construct the ColorDialog, use the call:

```
colordialog = QColorDialog(parent)
```

The *parent* argument supplied indicates the widget (i.e. window) which owns the ColorDialog.

49.2 Methods

Display of the ColorDialog is done using the call:

```
colordialog.open()
```

To obtain the colour information from the dialog use:

```
colordialog.selectedColor()
```

The current colour displayed on the dialog can also be retrieved via:

```
colordialog.currentColor()
```

Alternatively, it can be set programatically with:

```
colordialog.setCurrentColor(color)
```

The *color* parameter should be set to an appropriate *Color* object.

Options configuring the ColorDialog can be set using:

```
colordialog.setOption(option, setting)
```

The *setting* parameter should be set to a Boolean value indicating whether the option is enabled or not. The *option* value can be set to any of the following:

- `QColorDialog.ShowAlphaChannel` - show transparency setting widget.

- `QColorDialog.NoButtons` - do not show OK and Cancel buttons on dialog.
- `QColorDialog.DontUseNativeDialog` - use the Qt default colour dialog.

The options in use can be retrieved from the ColorDialog by calling:

```
colordialog.options()
```

49.3 Example

Below is an example of a ColorDialog:

Download: `ColorDialog`

CHAPTER

FIFTY

LISTWIDGET

A ListWidget is a simple list widget which provides an easy way to display a number of items, of which one or more can be selected.

50.1 Constructor

Constructing the ListWidget is done by:

```
listwidget = QListWidget()
```

50.2 Methods

Item can be added to the ListWidget via several different methods:

```
listwidget.addItem(text)
listwidget.addItem(item)
listwidget.addItems(text, text, ...)
listwidget.insertItem(row, text)
listwidget.insertItem(row, item)
listwidget.insertItems(row, text, text, ...)
```

The first method takes a string of text as the parameter and adds it to the list. The second method takes a *ListWidgetItem* as a parameter to display. The final method takes several strings of text and adds each one as a single item to the ListWidget. The 'insert' methods work in the same way, with the additional integer value indicating the row at which the item is to be added.

Removal of items from the list is done by passing the *ListWidgetItem* object in the method:

```
listwidget.removeItemWidget(item)
```

If editing of items is permitted, the item to edit can be declared via:

```
listwidget.editItem(item)
```

The number of items currently in the list can be retrieved with:

```
count = listwidget.count()
```

To enable sorting of the items in the list call:

```
listwidget.sortingEnabled(enabled)
```

The direction of the sort can also be configured by:

```
listwidget.sortItems(order)
```

The *order* parameter should be set to one of:

- `Qt.AscendingOrder`
- `Qt.DescendingOrder`

Items in the list can be programatically selected via their row number with:

```
listwidget.setCurrentRow(row)
```

The *row* value should be the number identifying the item in the list, with `0` indicating the first item should be selected.

The current row selected can also be retrieved:

```
listwidget.currentRow()
```

Items can be selected based on their ListWidgetItem object via:

```
listwidget.setCurrentItem(item)
```

Alternatively, the current item can be fetched when selected by the method:

```
listwidget.currentItem()
```

50.3 Example

Below is an example of a ListWidget:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        self.listwidget = QListWidget()
        self.listwidget.insertItem(0, "Orange")
        self.listwidget.insertItem(1, "Rose")
        self.listwidget.insertItem(2, "Brown")
        self.listwidget.insertItem(3, "Mauve")
        self.listwidget.insertItem(4, "Sapphire")
        self.listwidget.clicked.connect(self.listview_clicked)
        layout.addWidget(self.listwidget)

    def listview_clicked(self, qmodelindex):
        item = self.listwidget.currentItem()
        print(item.text())

app = QApplication(sys.argv)
```

```
screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `ListWidget`

CHAPTER

FIFTYONE

LISTWIDGETITEM

The ListWidgetItem is used to provide an item for use within the *ListWidget*. Each item holds several pieces of information and displays the items as per the information.

51.1 Constructor

In many cases, the ListWidgetItem will not need to be constructed manually as one is created for each item added to a ListWidget. If required however, it is constructed via:

```
listwidgetitem = QListWidgetItem()
```

51.2 Methods

The textual string of the item can be set using:

```
listwidgetitem.setText(text)
```

It can also be retrieved via:

```
listwidgetitem.text()
```

The item can be hidden from the viewing widget with the call:

```
listwidgetitem.setHidden(hidden)
```

When *hidden* is `True`, the item will not be visible to the user.

To check whether an item is hidden from view call:

```
listwidgetitem.isHidden()
```

An exact copy of the ListWidgetItem including all properties set can be made with:

```
listwidgetitem.clone()
```

The ListWidget which contains the item can be fetched if required by:

```
listwidgetitem.listWidget()
```

CHAPTER

FIFTYTWO

TABLEWIDGET

The TableWidget is a complex widget providing rows and columns of information in a grid-like format. It supports a variety of features such as row and column headers, multiple selections, and sorting functionality.

52.1 Constructor

The TableWidget is constructed using the call:

```
tablewidget = QTableWidget()
```

52.2 Methods

The number of rows and columns to be displayed by the TableWidget must be declared with the methods:

```
tablewidget.setRowCount(count)
tablewidget.setColumnCount(count)
```

The number of rows and columns can be obtained from the TableWidget:

```
tablewidget.rowCount()
tablewidget.columnCount()
```

Hiding individual columns can be done with:

```
tablewidget.setColumnHiden(column, hidden)
```

The *column* value indicates the positional column value, with the first column identified by `0`. The *hidden* value when set to `True` will hide the column from view.

To configured whether the TableWidget grid lines are toggled on or off use:

```
tablewidget.setShowGrid(show_grid)
```

When *show_grid* is set to `True` each cell will have a box around it to make identifying rows and columns easier.

The row and column headers are contained as separate objects which are obtained via:

```
tablewidget.horizontalHeader()
tablewidget.verticalHeader()
```

By default, the TableWidget allows multiple rows to be selected. This can be configured using:

```
tablewidget.setSelectionMode(mode)
```

The *mode* parameter should be set to one of:

- `QAbstractItemView.NoSelection`
- `QAbstractItemView.SingleSelection`
- `QAbstractItemView.ContiguousSelection`
- `QAbstractItemView.ExtendedSelection`
- `QAbstractItemView.MultiSelection`

Also default is the ability to edit the contents of a cell when selected. This is set with the method:

```
tablewidget.setEditTriggers(triggers)
```

The *triggers* value can be set to one of the following:

- `QAbstractItemView.NoEditTriggers` - no editing possible.
- `QAbstractItemView.CurrentChanged` - start editing when the current item changes.
- `QAbstractItemView.DoubleClicked` - start editing when item is double-clicked.
- `QAbstractItemView.SelectedClicked` - start editing when clicking an already selected item.
- `QAbstractItemView.EditKeyPressed` - start editing when platform edit key is pressed.
- `QAbstractItemView.AnyKeyPressed` - start editing when any key is pressed.
- `QAbstractItemView.AllEditTriggers` - start editing when any of the above are activated.

52.3 Example

Below is an example of a TableWidget:

Download: `TableWidget`

CHAPTER

FIFTYTHREE

COLUMNVIEW

A ColumnView widget is similar to a *ListWidget*, but it typically contains data which has subitems. Each subitem is placed horizontally within a new ListWidget. This method of displaying information is sometimes called a cascading list.

53.1 Constructor

The constructor for the ColumnView is:

```
columnview = QColumnView()
```

53.2 Methods

To configure whether display grips are visible, allowing each column to be resized, use:

```
columnview.setResizeGripsVisible(visible)
```

The width of each column can be defined in pixels via:

```
columnview.setColumnWidgets(widths)
```

The *widths* parameter should be set to a list of sizes; one for each column. If the list does not contain enough values for each column, the columns with no value specified will not be modified. If the list contains more values than columns, the extra values will be used for any columns added later.

The widths of each column is defined using:

```
columnview.setColumnWidths(width)
```

The *width* parameter should be a passed list with a value for each column defining the width in pixels.

Returning the column widths from the ColumnView is done using the method:

```
columnview.columnWidths()
```

CHAPTER

FIFTYFOUR

SCROLLAREA

A ScrollArea widget provides a container for another widget to be placed, providing scrolling in both vertical and horizontal directions when the child is larger than the space allocated.

The ScrollArea automatically provides *ScrollBar* objects and is preferred in most cases when scrolling must be provided.

54.1 Constructor

Construction of the ScrollArea is made using:

```
scrollarea = QScrollArea()
```

54.2 Methods

Widgets are added to the ScrollArea container using:

```
scrollarea.setWidget(widget)
```

The widget assigned to the ScrollArea can be retrieved with:

```
scrollarea.widget()
```

The added widget can be positioned within the area via:

```
scrollarea.setAlignment(alignment)
```

Set the *alignment* value to one of the following:

- `Qt.AlignLeft`
- `Qt.AlignRight`
- `Qt.AlignTop`
- `Qt.AlignBottom`
- `Qt.AlignHCenter`
- `Qt.AlignVCenter`

The child widget can be resized within the ScrollArea via:

```
scrollarea.setWidgetResizable(resizable)
```

When *resizable* is set to `True`, the ScrollArea automatically resizes the widget to try and avoid scroll bars and take advantage of extra space. If set to `False`, the default widget size is honoured.

54.3 Example

Below is an example of a ScrollArea:

Download: `ScrollArea`

CHAPTER

FIFTYFIVE

PLAINTEXTEDIT

The PlainTextEdit widget is optimised to display plain text content.

If the application is to display formatted text, the *TextEdit* widget should be used.

55.1 Constructor

The PlainTextEdit widget is constructed by using:

```
plaintextedit = QPlainTextEdit()
```

55.2 Methods

Text is inserted into the PlainTextEdit by either of the following methods:

```
plaintextedit.appendPlainText(text)
plaintextedit.insertPlainText(text)
```

The `.appendPlainText()` method adds the new text to the end of the current text block whereas the `.insertPlainText()` method adds the text at the cursor position.

By default, the text in the PlainTextEdit can be modified by the user. It can however be used as a read-only widget with:

```
plaintextedit.setReadOnly(read_only)
```

When *read_only* is set to `True`, the user will only be able to navigate through the text.

The read-only state of the widget can also be retrieved using:

```
plaintextedit.isReadOnly()
```

Placeholder text can be placed into the PlainTextEdit with:

```
plaintextedit.setPlaceholderText(text)
```

The *text* specified will only be shown in the widget when there is no text loaded.

The title of the document can be set via:

```
plaintextedit.setDocumentTitle(title)
```

Retrieval of the title string is also done with:

```
plaintextedit.documentTitle()
```

The text held by the PlainTextEdit can also be line wrapped if required:

```
plaintextedit.setLineWrapMode(mode)
```

The *mode* value should be set to one of the following:

- `QPlainTextEdit.NoWrap` - do not wrap the text.
- `QPlainTextEdit.WidgetWidth` - wrap text at width of PlainTextEdit.

Word wrapping is also enabled separately:

```
plaintextedit.setWordWrapMode(mode)
```

The *mode* value in this case should be set to:

- `QTextOption.NoWrap` - text is not wrapped.
- `QTextOption.WordWrap` - wrap text at end of words.
- `QTextOption.ManualWrap` - same as the `NoWrap` constant.
- `QTextOption.WrapAnywhere` - wrap text anywhere, even in the middle of a word if required.
- `QTextOption.WrapAtWordBoundaryOrAnywhere` - wrap at end of a word, or anywhere if there is no other option.

By default, any text entered into the PlainTextEdit will be inserted. Existing text can be overwritten instead via:

```
plaintextedit.setOverwriteMode(overwrite)
```

Undo and redo support is enabled on a PlainTextEdit. This can be turned off if not required using:

```
plaintextedit.setUndoRedoEnabled(enable)
```

55.3 Example

Below is an example of a PlainTextEdit:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        plaintextedit = QPlainTextEdit()
        plaintextedit.setPlaceholderText("This is some placeholder text.")
        layout.addWidget(plaintextedit, 0, 0)

app = QApplication(sys.argv)

screen = Window()
```

```
screen.show()

sys.exit(app.exec_())
```

Download: `PlainTextEdit`

CHAPTER

FIFTYSIX

TEXTEDIT

The TextEdit widget is a powerful text display widget, with the ability to display both plain text and formatted text. It can handle paragraphs, images, tables, and lists, with the rich text display ability powered by HTML markup.

Smaller amounts of text are probably best being displayed using the *Label* widget, or alternatively, the *LineEdit* widget if the user should be able to manipulate the text. Alternatively, if the application only handles plain text content, it is better to use *PlainTextEdit*.

56.1 Constructor

The TextEdit widget is constructed by using:

```
textedit = QTextEdit()
```

56.2 Methods

Text content can be added using a number of methods:

```
textedit.append(text)
textedit.insertHtml(text)
textedit.insertPlainText(text)
textedit.setText(text)
textedit.setHtml(text)
textedit.setPlainText(text)
```

The `append()` method adds text to the position of the cursor. Alternatively, the `insertHtml()` and `insertPlainText()` allows text to be added either with rich text or plain text. The `setText()`, `setHtml()` and `setPlainText()` methods replace the existing content of the TextEdit with the new text.

Content from the TextEdit can be retrieved with the calls:

```
textedit.toHtml()
textedit.toPlainText()
```

All text within the TextEdit can be cleared using:

```
textedit.clear()
```

In some circumstances, the TextEdit may only accept or display plain text. This is set via:

```
textedit.setAcceptRichText(rich_text)
```

To ensure that a user can not change text held in the TextEdit, call:

```
textedit.setReadOnly(read_only)
```

The read-only state of the TextEdit is fetchable via:

```
textedit.isReadOnly(read_only)
```

Placeholder text can be added to the TextEdit, which is displayed when no other text is added:

```
textedit.setPlaceholderText(text)
```

By default, any text added to the TextEdit will be inserted. Existing content can instead be overwritten via:

```
textedit.setOverwriteMode(overwrite)
```

TextEdit widgets automatically support undo and redo actions. These can be called with:

```
textedit.undo()
textedit.redo()
```

If undo/redo support is not required, this can be turned of using the method:

```
textedit.setUndoRedoEnabled(enable)
```

Words within the TextEdit default to wrapping at the end of a word. This is configured by:

```
textedit.setLineWrapMode(mode)
```

The *mode* should be set to one of:

- `QTextEdit.NoWrap` - do not wrap the text.
- `QTextEdit.WidgetWidth` - wrap text at width of TextEdit.

The mode in use when wrapping words can also be configured by the method:

```
textedit.setWordWrapMode(mode)
```

The *mode* value in this case should be defined to one of:

- `QTextOption.NoWrap` - text is not wrapped.
- `QTextOption.WordWrap` - wrap text at end of words.
- `QTextOption.ManualWrap` - same as the `NoWrap` constant.
- `QTextOption.WrapAnywhere` - wrap text anywhere, even in the middle of a word if required.
- `QTextOption.WrapAtWordBoundaryOrAnywhere` - wrap at end of a word, or anywhere if there is no other option.

56.3 Example

Below is an example of a TextEdit:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
```

```
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        textedit = QTextEdit()
        textedit.setPlaceholderText("This is some placeholder text.")
        layout.addWidget(textedit, 0, 0)

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `TextEdit`

CHAPTER

FIFTYSEVEN

SPLASHSCREEN

A SplashScreen is commonly used by large applications which can take some time to startup. The SplashScreen usually provides the name and logo of the application, and occasionally a *ProgressBar* to indicate the progress made in starting the program.

It is recommended to only use a SplashScreen where required.

57.1 Constructor

Construction of the SplashScreen is made using the call:

```
splashscreen = QSplashScreen(pixmap)
```

The *pixmap* parameter should be set to an appropriate `pixmap` image which will be displayed on the SplashScreen.

57.2 Methods

The SplashScreen can be displayed when required with:

```
splashscreen.show()
```

A SplashScreen is able to be closed automatically when the main window is shown with:

```
splashscreen.finish(window)
```

The *window* argument should be set to the main window which the SplashScreen will wait for.

Displaying of a message on the SplashScreen is able to be done via the method:

```
splashscreen.showMessage(message)
```

A message can also be cleared from display via:

```
splashscreen.clearMessage()
```

57.3 Example

Below is an example of a SplashScreen:

Download: `SplashScreen`

CHAPTER

FIFTYEIGHT

MESSAGEBOX

The MessageBox widget is a subclass of the *Dialog* object. It is tailored for displaying short messages to the user such as information or errors, but can also be used to ask simple questions.

58.1 Constructor

The MessageBox widget is constructed using:

```
messagebox = QMessageBox()
```

58.2 Methods

The text on the MessageBox can be set with:

```
messagebox.setText(text)
```

If a more detailed description of the message is required, such as a portion of a log file, this can be displayed using:

```
messagebox.setInformativeText(text)
```

Standard buttons are Qt provided buttons which can easily be added to the MessageBox without the user having to create each one manually. These can be set via:

```
messagebox.setStandardButtons(buttons)
```

The standard buttons supported are:

- `QMessageBox.Ok`
- `QMessageBox.Open`
- `QMessageBox.Save`
- `QMessageBox.Cancel`
- `QMessageBox.Close`
- `QMessageBox.Discard`
- `QMessageBox.Apply`
- `QMessageBox.Reset`
- `QMessageBox.RestoreDefaults`
- `QMessageBox.Help`

- QMessageBox.SaveAll
- QMessageBox.Yes
- QMessageBox.YesToAll
- QMessageBox.No
- QMessageBox.NoToAll
- QMessageBox.Abort
- QMessageBox.Retry
- QMessageBox.Ignore

A user can add and remove extra buttons manually with:

```
messagebox.addButton(button)
messagebox.removeButton(button)
```

In both cases, the *button* object points to the button object (such as a *PushButton*) to be added or removed.

Icons can be added to the MessageBox to further indicate the purpose of the content:

```
messagebox.setIcon(icon)
```

The *icon* parameter should be set to:

- QMessageBox.NoIcon
- QMessageBox.Question
- QMessageBox.Information
- QMessageBox.Warning
- QMessageBox.Critical

58.3 Example

Below is an example of a MessageBox:

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

from PyQt5.QtWidgets import *
import sys

class MessageBox(QMessageBox):
    def __init__(self):
        QMessageBox.__init__(self)
        self.setText("This is a MessageBox, typically used to convey short messages to the user.")
        self.setInformativeText("Informative text provides more space to explain the message purpose.
        self.setIcon(QMessageBox.Information)
        self.setStandardButtons(QMessageBox.Close)

app = QApplication(sys.argv)

screen = MessageBox()
screen.show()

sys.exit(app.exec_())
```

Download: `MessageBox`

CHAPTER

FIFTYNINE

WIZARD

A Wizard is a helper widget which allows for paginated display of information which the user can progress through. They are commonly used for setup of new programs or building up information prior to an action.

59.1 Constructor

The Wizard is constructed with the statement:

```
wizard = QWizard()
```

59.2 Methods

A page can be added to the Wizard via the two methods:

```
wizard.addPage(page)
wizard.setPage(number, page)
```

The *page* parameter should be the *WizardPage* object which is to be added. The *number* indicates the position at which the page should be added.

Pages can also be removed by specifying the page number:

```
wizard.removePage(number)
```

The title used on the page can be set with:

```
wizard.setTitle(title)
```

The page object for a given page number can be retrieved using:

```
wizard.page(number)
```

The current page object and number can be retrieved with the calls:

```
wizard.currentPage()
wizard.currentId()
```

A check can be made on whether a user has visited a particular page via:

```
wizard.hasVisitedPage(number)
```

Alternatively, a list of visited pages can be obtained in list form with:

```
wizard.visitedPages()
```

The operation of the page movement can be done programmatically by:

```
wizard.back()
wizard.next()
wizard.restart()
```

The `.back()` and `.next()` methods will take the user back to the previous page on forward to the next page. The `.restart()` call takes the user back to the first page.

59.3 Example

Below is an example of a Wizard:

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

from PyQt5.QtWidgets import *
import sys

class Window(QWidget):
    def __init__(self):
        QWidget.__init__(self)

        layout = QGridLayout()
        self.setLayout(layout)

        button = QPushButton("Launch")
        button.clicked.connect(self.on_button_clicked)
        layout.addWidget(button)

        self.wizard = QWizard()

    def on_button_clicked(self):
        self.wizard.open()

app = QApplication(sys.argv)

screen = Window()
screen.show()

sys.exit(app.exec_())
```

Download: `Wizard`

CHAPTER

SIXTY

WIZARDPAGE

A WizardPage is the object holding the page content for display in the *Wizard*.

60.1 Constructor

A WizardPage object can be constructed using:

```
wizardpage = QWizardPage()
```

60.2 Methods

The title and subtitle can be set on a page using the calls:

```
wizardpage.setTitle(title)
wizardpage.setSubTitle(subtitle)
```

A page can be set to be the final page with:

```
wizardpage.setFinalPage(final)
```

A commit page, which can be undone by clicking Back or Cancel can be set via:

```
wizardpage.setCommitPage(commit)
```

To check whether a page has been completed call:

```
wizardpage.isComplete()
```

Additional methods are available to check whether a page is either a commit or final page:

```
wizardpage.isCommitPage()
wizardpage.isFinalPage()
```

60.3 Example

The WizardPage example is a part of the Wizard widget example.

CHAPTER

SIXTYONE

CLIPBOARD

The Clipboard object provides access to the system clipboard, allowing data to be copied and pasted between applications.

Note: The Clipboard support differs across platforms, such as Windows not supporting primary selection unlike X11. Some features may behave differently or be entirely unsupported.

61.1 Constructor

Construction of the Clipboard object is made using:

```
clipboard = QClipboard()
```

61.2 Methods

Data is set on the Clipboard using a number of calls depending on the data type:

```
clipboard.setText(text, mode)
clipboard.setImage(image, mode)
clipboard.setPixmap(pixmap, mode)
clipboard.setMimeData(mimedata, mode)
```

The *mode* parameter controls which part of the Clipboard is used, and should be set to:

- `QClipboard.Clipboard` - store and retrieve from the global clipboard.
- `QClipboard.Selection` - store and retrieve from the mouse selection (X11 and others).
- `QClipboard.FindBuffer` - store and retrieve from the Find buffer (OS X).

Data is also retrievable from the Clipboard with:

```
clipboard.text(mode)
clipboard.image(mode)
clipboard.pixmap(mode)
clipboard.mimeData(mode)
```

The contents of the Clipboard can be cleared via the method:

```
clipboard.clear()
```

Objects:

CHAPTER

SIXTYTWO

COLOR

The Color object provides a way for Qt to represent colours. It supports RGB, CMYK, and HSV values, and is used by the *ColorDialog* to represent colours being displayed.

62.1 Constructor

The constructor for the Color object is:

```
color = QColor()
```

Once initialised, the Color object defaults to `0, 0, 0` RGB.

Alternatively, a colour can be defined on the object at constructed with:

```
color = QColor(red, green, blue)
```

The *red*, *green*, and *blue* values should be an integer value between `0` and `255`.

62.2 Methods

The colour values can be retrieved from the Color object with:

```
color.red()
color.blue()
color.green()
color.yellow()
color.black()
color.cyan()
color.magenta()
```

The colour values are settable post-construction via:

```
color.setRed(red)
color.setBlue(blue)
color.setGreen(green)
color.setYellow(yellow)
color.setBlack(black)
color.setCyan(cyan)
color.setMagenta(magenta)
```

Hue, saturation and value numbers can also be fetched from the Color object:

```
color.hue()
color.saturation()
color.value()
```

HSV numbers are also set with the methods:

```
color.setHue(hue)
color.setSaturation(saturation)
color.setValue(value)
```

The transparency of the colour is fetched if required via:

```
color.alpha()
```

Alpha transparency is set using:

```
color.setAlpha(alpha)
```

CHAPTER

SIXTYTHREE

ICON

THe Icon object represents an image typically used to represent an action. They are commonly used on menus or buttons in association with a specific task such as saving a document or finding a string of text.

63.1 Constructor

Construction of an empty Icon object is made by:

```
icon = QIcon()
```

Alternative constructors which allow the data to be loaded immediately are:

```
icon = QIcon(filename)
icon = QIcon(pixmap)
```

The *filename* parameter specifies the location from which to load an image. The *pixmap* argument points to a `pixmap` object which will be loaded into the Icon object.

63.2 Methods

An Icon can be set with an image via the methods:

```
icon.addFile(filename, size, mode, state)
icon.addPixmap(pixmap, mode, state)
```

The *filename* parameter points to the file to be loaded with the `.addFile()` method. Alternatively, the `.addPixmap()` method allows a `pixmap` object to be loaded. A *size* object allows the icon size to be specified using a `size` object. The *mode* value indicates the state of the icon and should be set to:

- `QIcon.Normal` - display as the icon is available, and the user is not interacting with it.
- `QIcon.Disabled` - display when the functionality of the icon is not allowed.
- `QIcon.Active` - display when the functionality of the icon is available, and the user it interacting with it (e.g. on mouseover).
- `QIcon.Selected` - display when the icon is selected.

A *state* parameter can also be defined as to whether the Icon object is on or off with:

- `QIcon.Off`
- `QIcon.On`

An Icon can also be checked for emptiness using:

```
icon.isNull()
```

Icons can also be swapped if required with:

```
icon.swap(icon)
```

The *icon* parameter should be set to another Icon object with which the values should be switched.

CHAPTER

SIXTYFOUR

DATE

The Date object provides an interface for handling dates, and is used by some widgets such as the *Calendar* to represent a date.

Note: Alternatives to the Date object includes the *DateTime*, allowing both times and dates to be stored and the *Time* object which is used only for time values.

64.1 Constructor

Construction of a Date object is made using:

```
date = QDate()
```

Alternatively, the year, month, and day, can be specified at construction time via:

```
date = QDate(year, month, day)
```

64.2 Methods

A date can also be set post-construction of the object with:

```
date.setDate(year, month, day)
```

To retrieve a date from the Date object call:

```
date.getDate()
```

The number corresponding to each of the year, month, and day values can be fetched individually by:

```
date.year()
date.month()
date.day()
```

The held year, month, and day values in the Date object can be incremented using the method:

```
date.addYears(years)
date.addMonths(months)
date.addDays(days)
```

All the above functions return a new Date object with the new incremented date held.

The day of the week and day of the year values can be obtained using:

```
date.dayOfWeek()
date.dayOfYear()
```

Returning the number of days in the currently set month or year is done via:

```
date.daysInMonth()
date.daysInYear()
```

The number of days until a particular day is reached can be found by passing a Date object using:

```
date.daysTo(date)
```

The validity of the current Date object can be checked with:

```
date.isValid()
```

Alternatively, the Date object can be checked to see if it has been set or not by:

```
date.isNull()
```

The day or month string can be obtained from the Date object using the methods:

```
date.longDayName(day)
date.longMonthName(month)
date.shortDayName(day)
date.shortMonthName(month)
```

The long form functions return "Monday" or "October", while the short form returns "Tue" or "Mar". The *day* or *month* parameter should be the number of the day or month to be returned.

To check whether a particular year is a leap year or not, use:

```
date.isLeapYear(year)
```

A Date object can be compared to another Date via the methods:

```
date.operator!=(date)
date.operator==(date)
date.operator>(date)
date.operator<(date)
date.operator>=(date)
date.operator<=(date)
```

CHAPTER

SIXTYFIVE

TIME

The Time object contains details pertaining data related to a clock, with it being able to store hours, minutes, seconds, and milliseconds.

Note: The Time object does not know about timezone or daylight savings time. If these are required, use the *DateTime* object instead.

65.1 Constructor

Construction of the Time object is made using:

```
time = QTime()
```

To construct a Time object with the values already set use:

```
time = QTime(hours, minutes, seconds, milliseconds)
```

65.2 Methods

The values on the Time can be set post-construction by using the method:

```
time.setHMS(hours, minutes, seconds, milliseconds)
```

To retrieve the values set call:

```
time.hour()
time.minute()
time.second()
time.msec()
```

The number of seconds or milliseconds from the currently set time can be retrieved via:

```
time.secsTo(time)
time.msecsTo(time)
```

The *time* parameter should be another Time object with the values to query.

Seconds or milliseconds can be added to the existing Time to create a new time object using the functions:

```
time.addSecs(seconds)
time.addMSecs(milliseconds)
```

To check whether the current time is valid call:

```
time.isValid()
```

The Time can also be checked to see whether any values have been set with:

```
time.isNull()
```

A second Time object can be compared using the operators:

```
time.operator==(time)
time.operator!=(time)
time.operator>(time)
time.operator<(time)
time.operator>=(time)
time.operator<=(time)
```

CHAPTER

SIXTYSIX

DATETIME

The DateTime object provides the ability to store both date and time information.

Note: The DateTime object can be replaced with the *Time* object if only representation of time is required. The *Date* object can also be used it only the date is to be handled.

66.1 Constructor

The constructor for the DateTime object is:

```
datetime = QDateTime()
```

66.2 Methods

CHAPTER

SIXTYSEVEN

DIR

The Dir object provides access to directories and their contents. It is used to manipulate paths, access information regardings those directories, and change the file system.

A Dir object can point to a path in absolute or relative form.

67.1 Constructor

The constructor for the Dir object is:

```
dir = QDir(path)
```

If the *path* parameter is not specified, the Dir object sets the path to the current working directory. Alternatively, another path can be specified.

67.2 Methods

To return the current directory, use:

```
dir.current()
```

An absolute or canonical paths to the directory can be fetched with:

```
dir.absolutePath()
dir.canonicalPath()
```

An absolute path can also be retrieved by specifying a path with:

```
dir.absoluteFilePath(filename)
```

The name of a set directory can be pulled from the object by:

```
dir.dirName()
```

The total number of directories and files within the specified directory is retrievable via:

```
dir.count()
```

Moving up through the directory structure is possible with the method:

```
dir.cdUp()
```

Alternatively, to change to a different directory call:

```
dir.cd(dirname)
```

The `cd()` method returns `True` if the directory exists and the method called successfully, otherwise `False` is returned.

A check can be performed on a passed filename with:

```
dir.exists(filename)
```

Removal of the defined file can be done via:

```
dir.remove(filename)
```

The `remove()` method returns `True` if the file is successfully removed, otherwise `False` is returned.

The Dir object can also be used to rename a file by:

```
dir.rename(oldname, newname)
```

Readability of the set file can be checked with:

```
dir.isReadable()
```

CHAPTER

SIXTYEIGHT

FILE

The File object provides an interface for reading and writing files. It supports handling of both text and binary files.

68.1 Constructor

Construction of the File object is made using:

```
file = QFile()
```

68.2 Methods

The file name which the File object points to is set with the call:

```
file.setFileName(filename)
```

Retrieval of the file name is done using:

```
file.fileName()
```

A file can be copied to a new location with:

```
file.copy(filename)
```

The *filename* parameter specifies the position the copied file will be created in.

Links can also be created as well via:

```
file.link(filename)
```

Renaming of the file name held by the File object is made by calling:

```
file.rename(filename)
```

Deletion of the file handled by the method:

```
file.remove()
```

Checking whether a file exists on the hard disk can be done with:

```
file.exists()
```

The method returns to `True` if the set file name exists, otherwise `False` is returned.