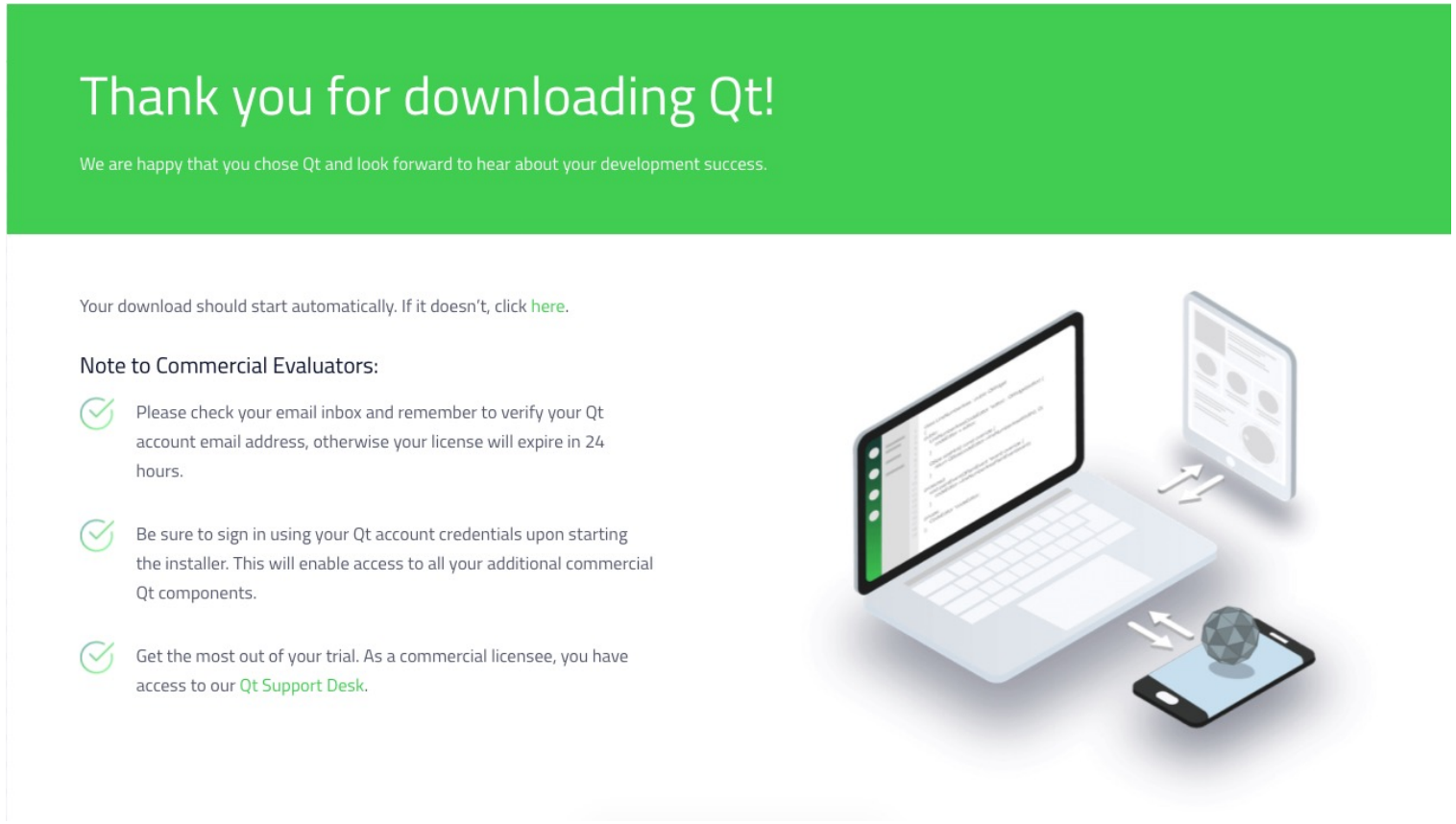
Lecture 13: PyQT

QT

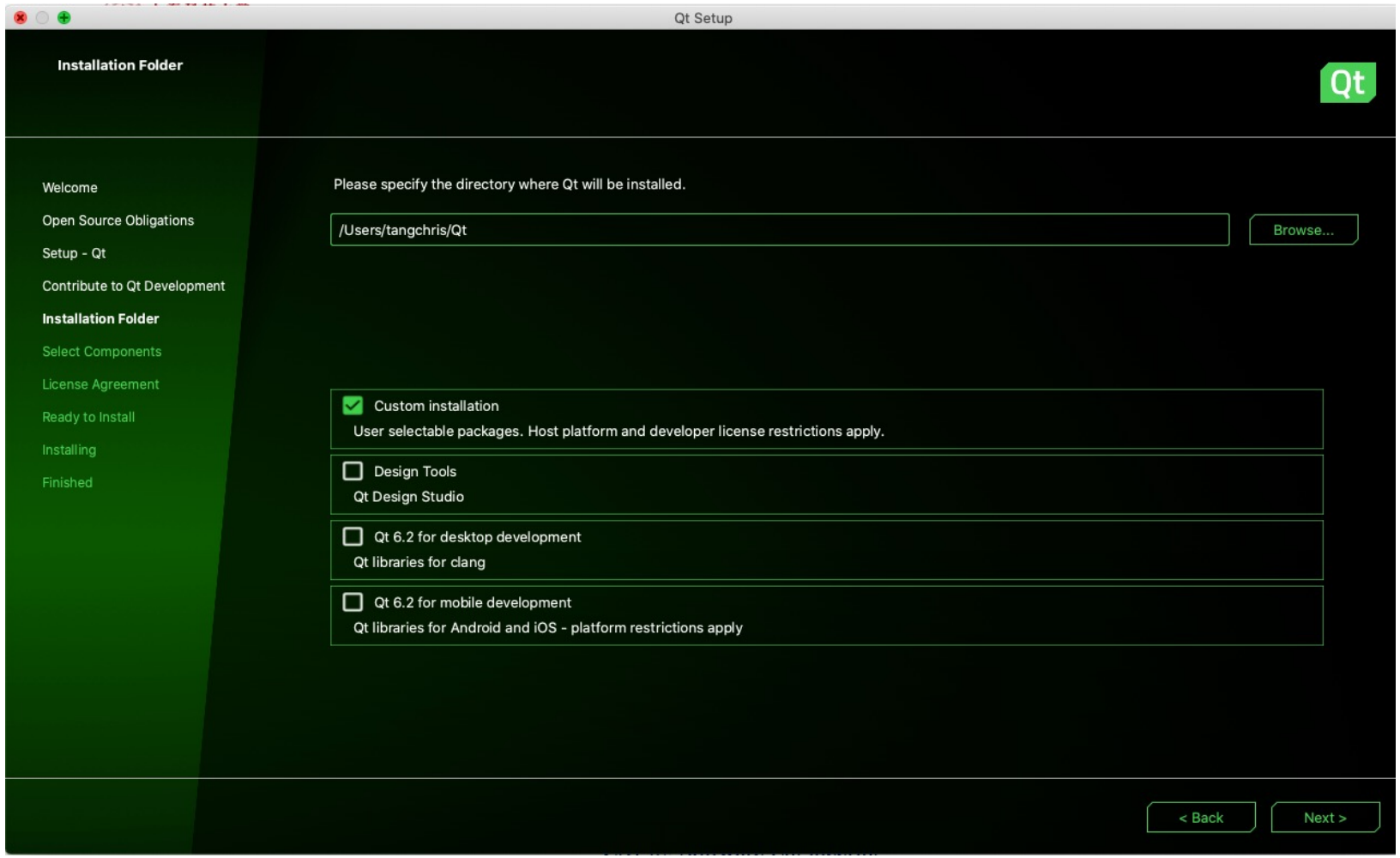
- https://www.qt.io/
- Qt is the faster, smarter way to create innovative devices, modern UIs & applications for multiple screens. Cross-platform software development at its best.

Download QT

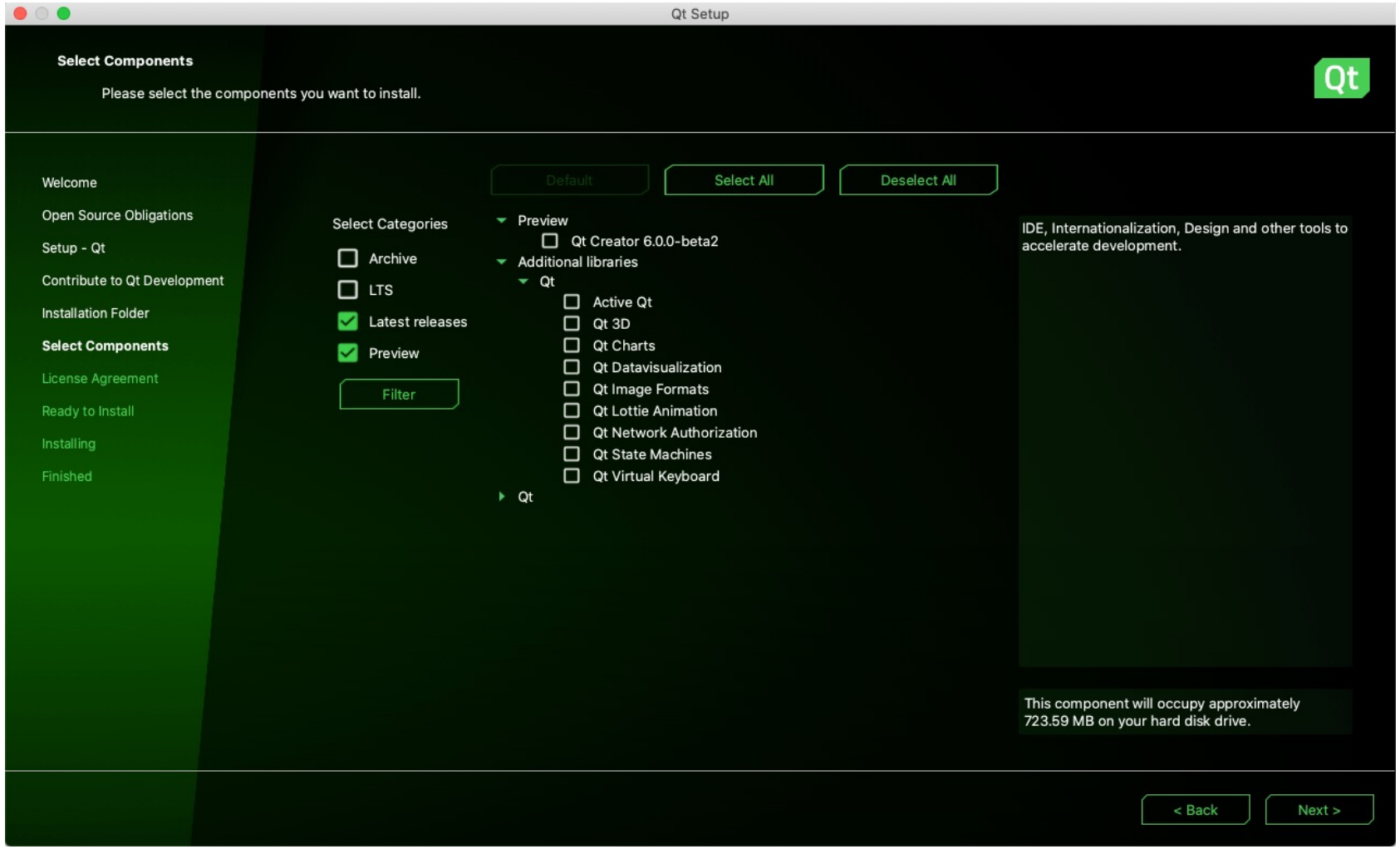
- https://www.qt.io/download-open-source



Install QT


Qt Setup
Installation Folder
Qt
Welcome
Open Source Obligations
Setup - Qt
Contribute to Qt Development
Installation Folder
Select Components
License Agreement
Ready to Install
Installing
Finished
Please specify the directory where Qt will be installed.
/Users/tangchris/Qt
Browse...
Custom installation
User selectable packages. Host platform and developer license restrictions apply.
Design Tools
Qt Design Studio
Qt 6.2 for desktop development
Qt libraries for clang
Qt 6.2 for mobile development
Qt libraries for Android and iOS - platform restrictions apply
< Back
Next >

Install QT (2)


Qt Setup
Select Components
Please select the components you want to install.
Qt
Welcome
Open Source Obligations
Setup - Qt
Contribute to Qt Development
Installation Folder
Select Components
License Agreement
Ready to Install
Installing
Finished
Default
Select All
Deselect All
Select Categories
Archive
LTS
Latest releases
Preview
Filter
Preview
Qt Creator 6.0.0-beta2
Additional libraries
Qt
Active Qt
Qt 3D
Qt Charts
Qt Datavisualization
Qt Image Formats
Qt Lottie Animation
Qt Network Authorization
Qt State Machines
Qt Virtual Keyboard
Qt
IDE, Internationalization, Design and other tools to accelerate development.
This component will occupy approximately 723.59 MB on your hard disk drive.
< Back
Next >

PyQT

- PyQT: https://riverbankcomputing.com/software/pyqt/
- PyQT is a set of Python bindings for The QT Company's QT application framework and runs on all platforms supported by QT including Windows, macOS, Linux, iOS and Android;
- PyQT 6 supports QT v6, PyQT 5 supports QT v5, and
- PyQT 4 supports QT v4.

Install PyQT

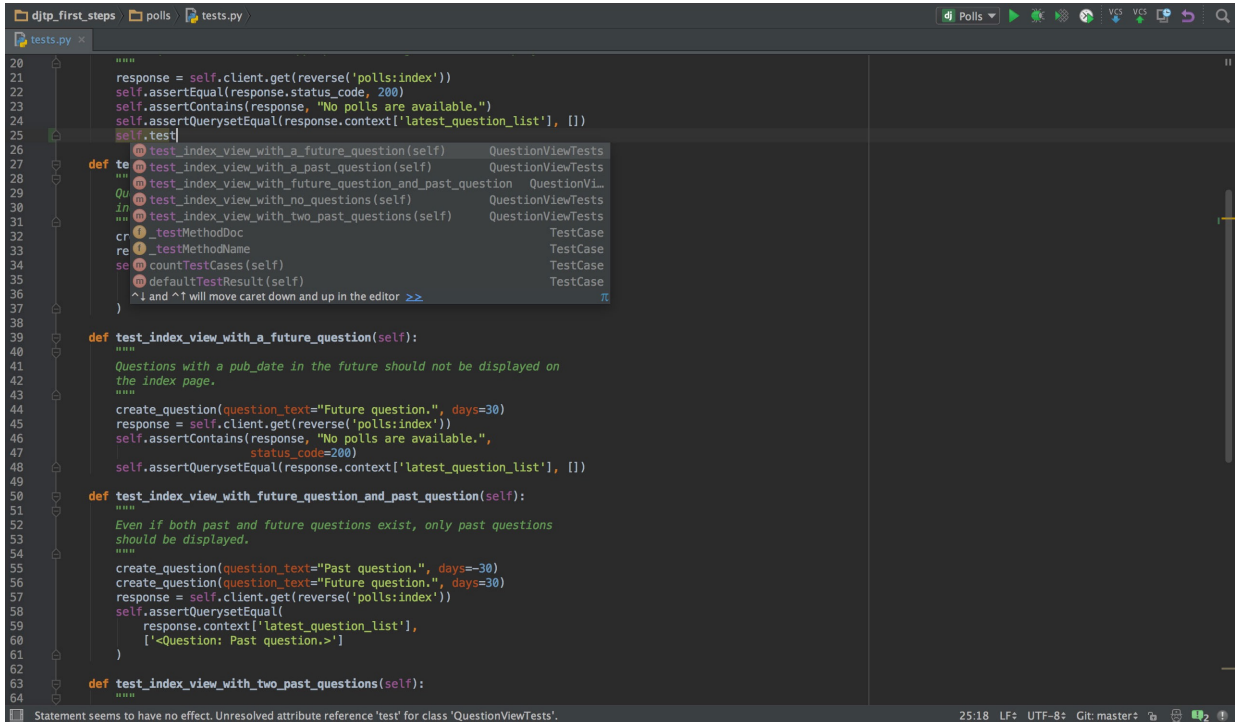
- PyQT 6
  - pip install PyQt6
- PyQT 5
  - Pip install PyQt5

PyQT API Reference

- The classes in PyQT
- https://www.riverbankcomputing.com/static/Docs/PyQt6/sip-classes.html

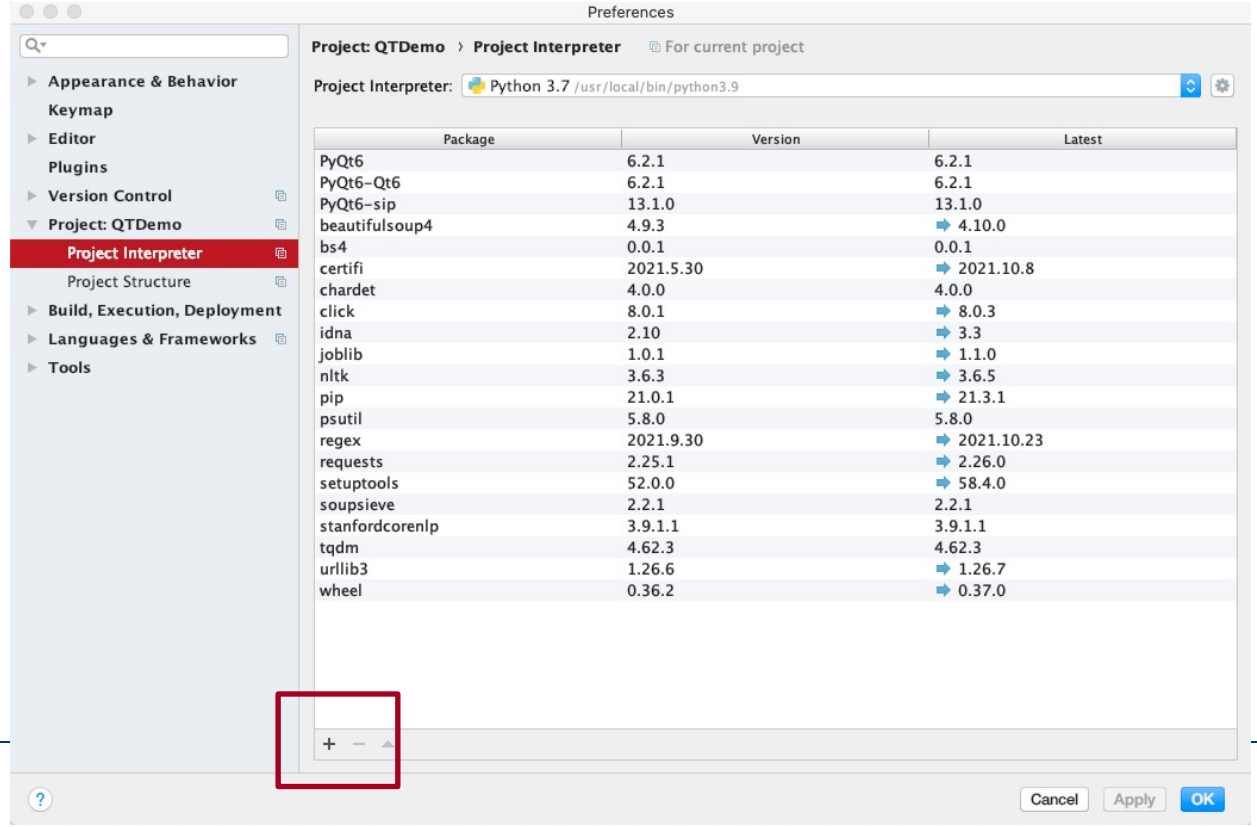
PyCharm IDE

- https://www.jetbrains.com/pycharm/
- PyCharm IDE for Academic/Education



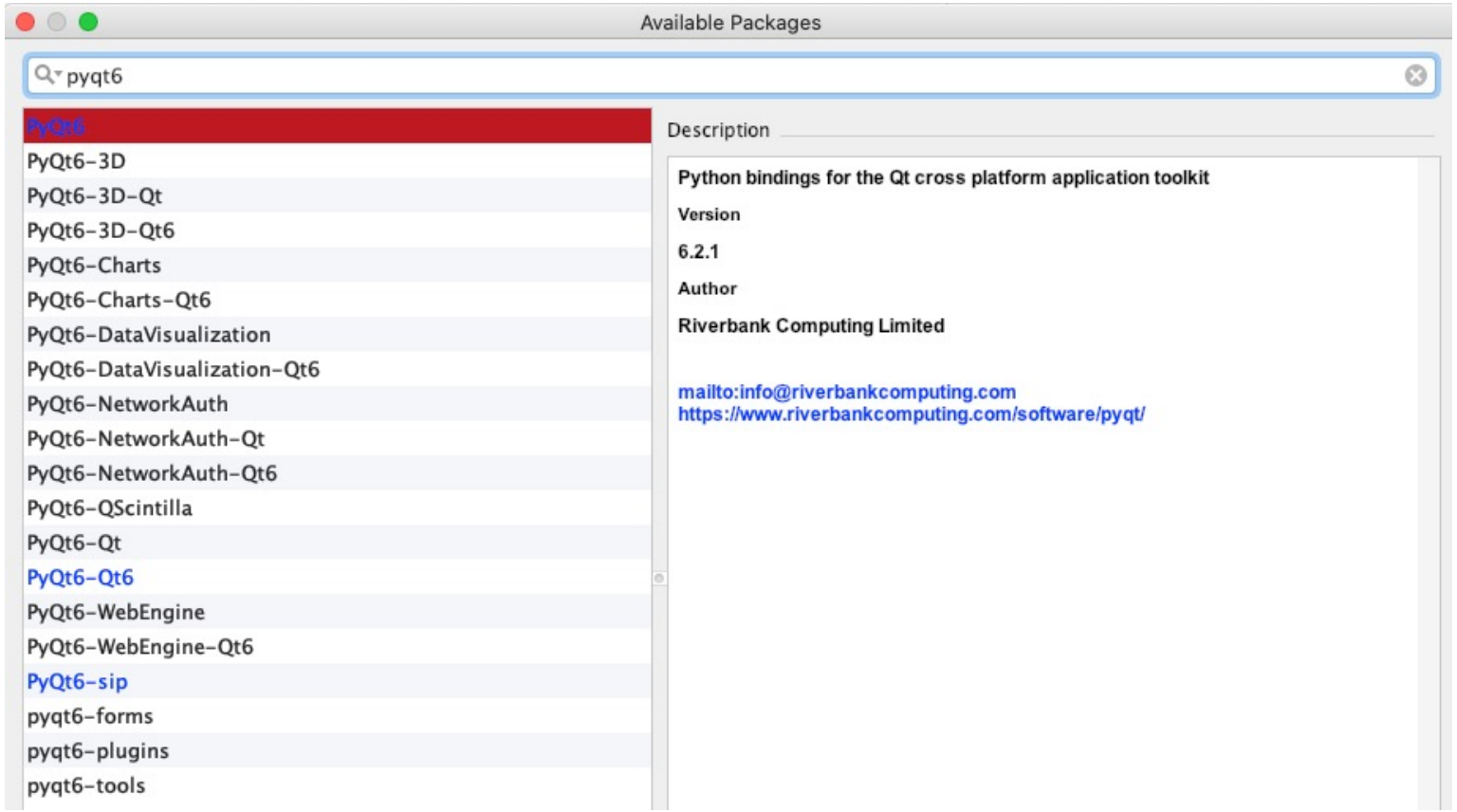
PyCharm IDE (2)

- Import the pyqt6 in PyCharm
- PyCharm -> Preference -> Project Interpreter
- Click “+”



PyCharm IDE (3)

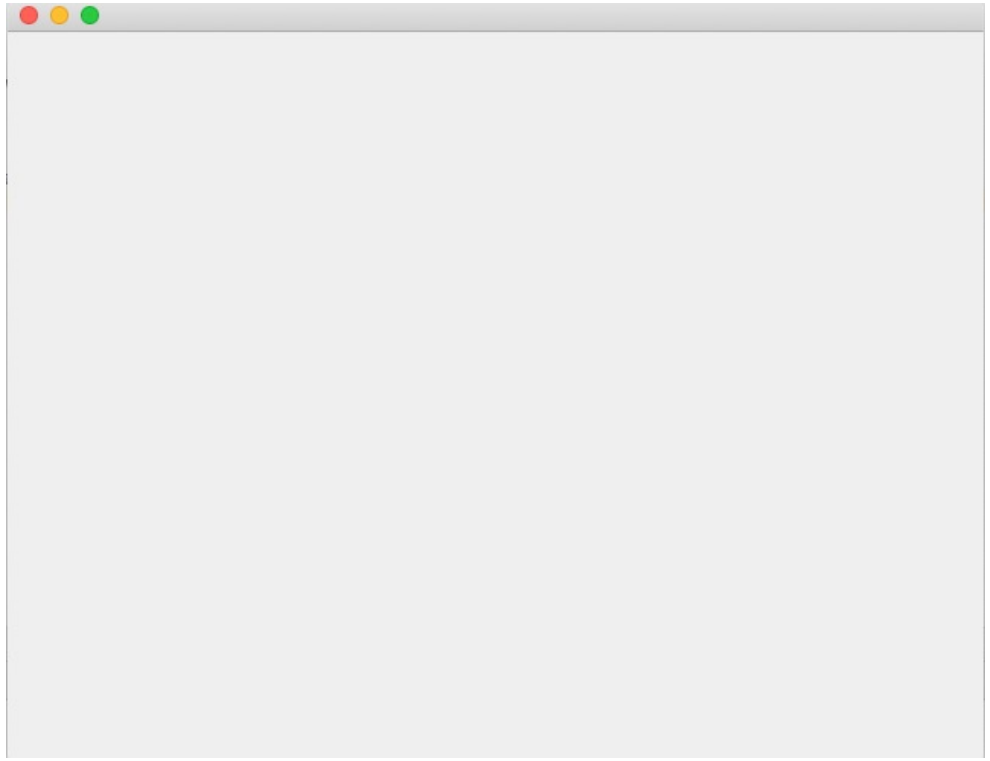
- Search pyqt6 and click install
- Save



A startup demo

- **from** PyQt6.QtWidgets **import** QApplication, QWidget
  import sys

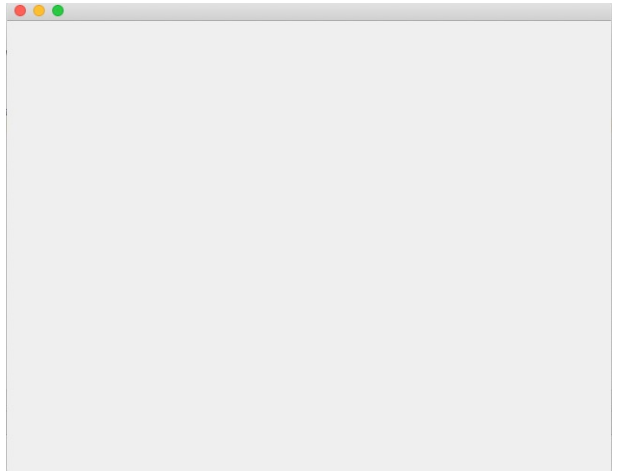
  if __name__ == '__main__':
  app = QApplication(sys.argv)
  window = QWidget()
  window.show()
  app.exec()



A startup demo (2)

- **from** PyQt6.QtWidgets **import** QApplication, QWidget
  import sys

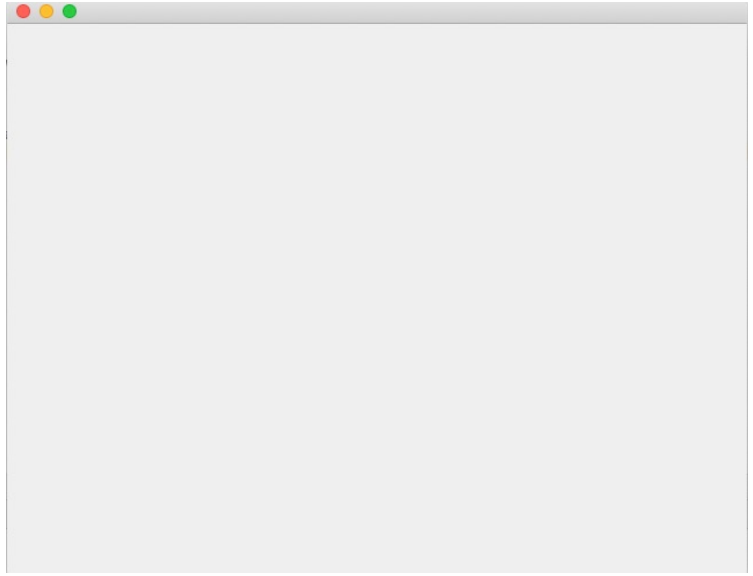
  First, we import the PyQt6 classes that we need for the application.
- Here we're importing QApplication, the application handler and QWidget, a basic empty GUI widget, both from the QtWidgets module.



A startup demo (3)

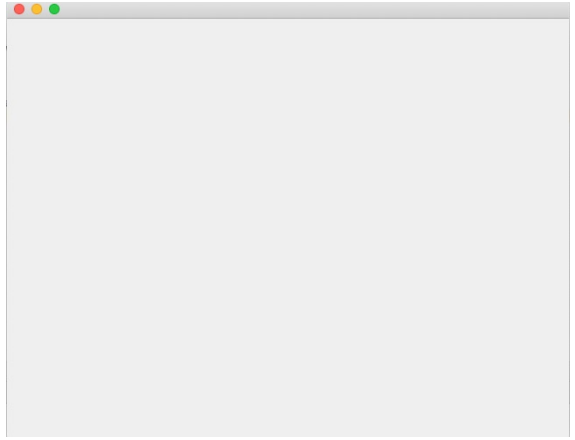
- **if** __name__ == '**__main__**':
  app = QApplication(sys.argv)

- You need one (and only one) QApplication instance per application.
  - (1) Pass in sys.argv to allow command line arguments for your app.
  - (2) If you know you won't use command line arguments QApplication([]) works too.



A startup demo (4)

- **if** __name__ == '**__main__**':
  app = QApplication(sys.argv)
  window = QWidget()
  window.show()
  app.exec()

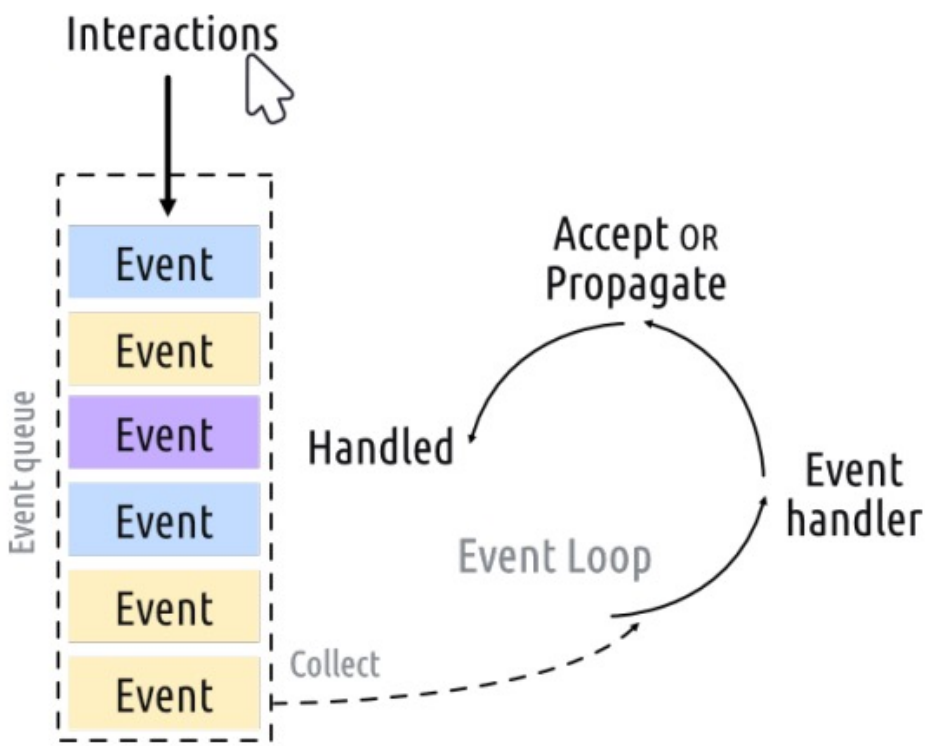


- Next we create an instance of a QWidget using the variable name window
- Widgets without a parent are invisible by default. So, after creating the window object, we must always call .show() to make it visible

- Finally, we call app.exec_() to start up the event loop

Event Loop

- The core of every Qt Applications is the QApplication class.
- Every application needs one — and only one — QApplication object to function.
- This object holds the event loop of your application — the core loop which governs all user interaction with the GUI



Event Loop (2)

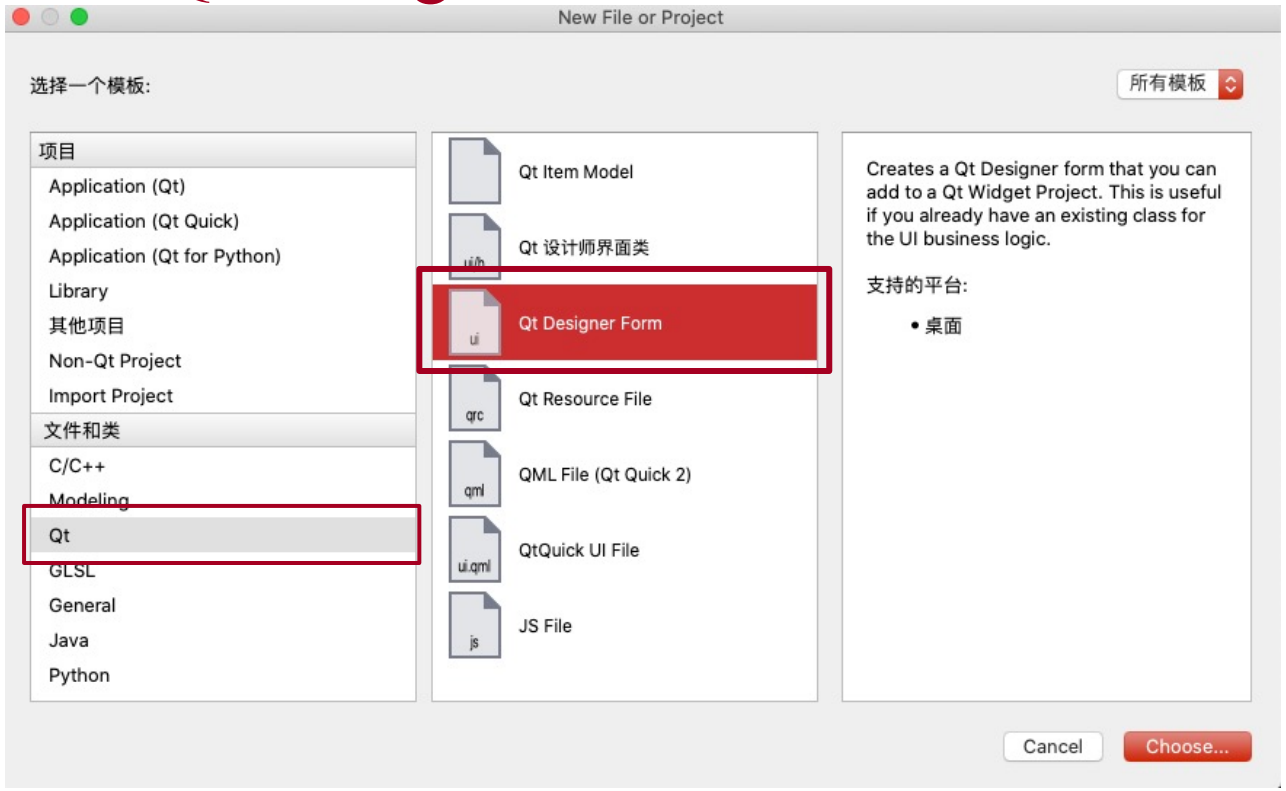
- Each interaction with your application — whether a press of a key, click of a mouse, or mouse movement — generates an event which is placed on the event queue.
- In the event loop, the queue is checked on each iteration and if a waiting event is found, the event and control is passed to the specific event handler for the event.
- The event handler deals with the event, then passes control back to the event loop to wait for more events. There is only one running event loop per application.

UI Designer

- Open QT Creator application (* find it in QT's installation directory)

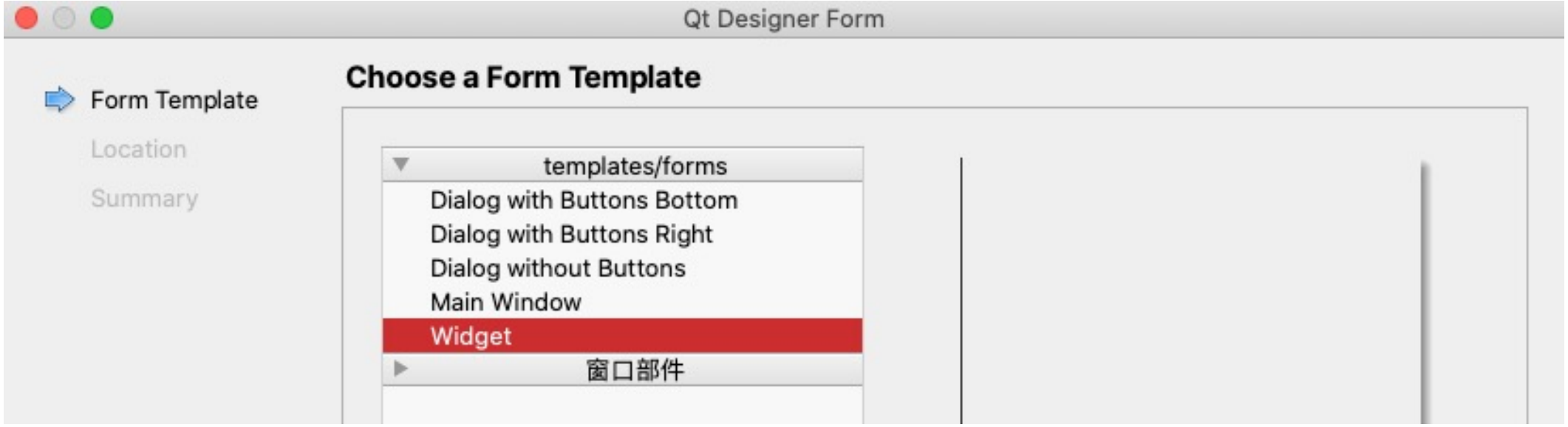
UI Designer (2)

- File -> New File or Project
- Choose QT → QT Designer Form

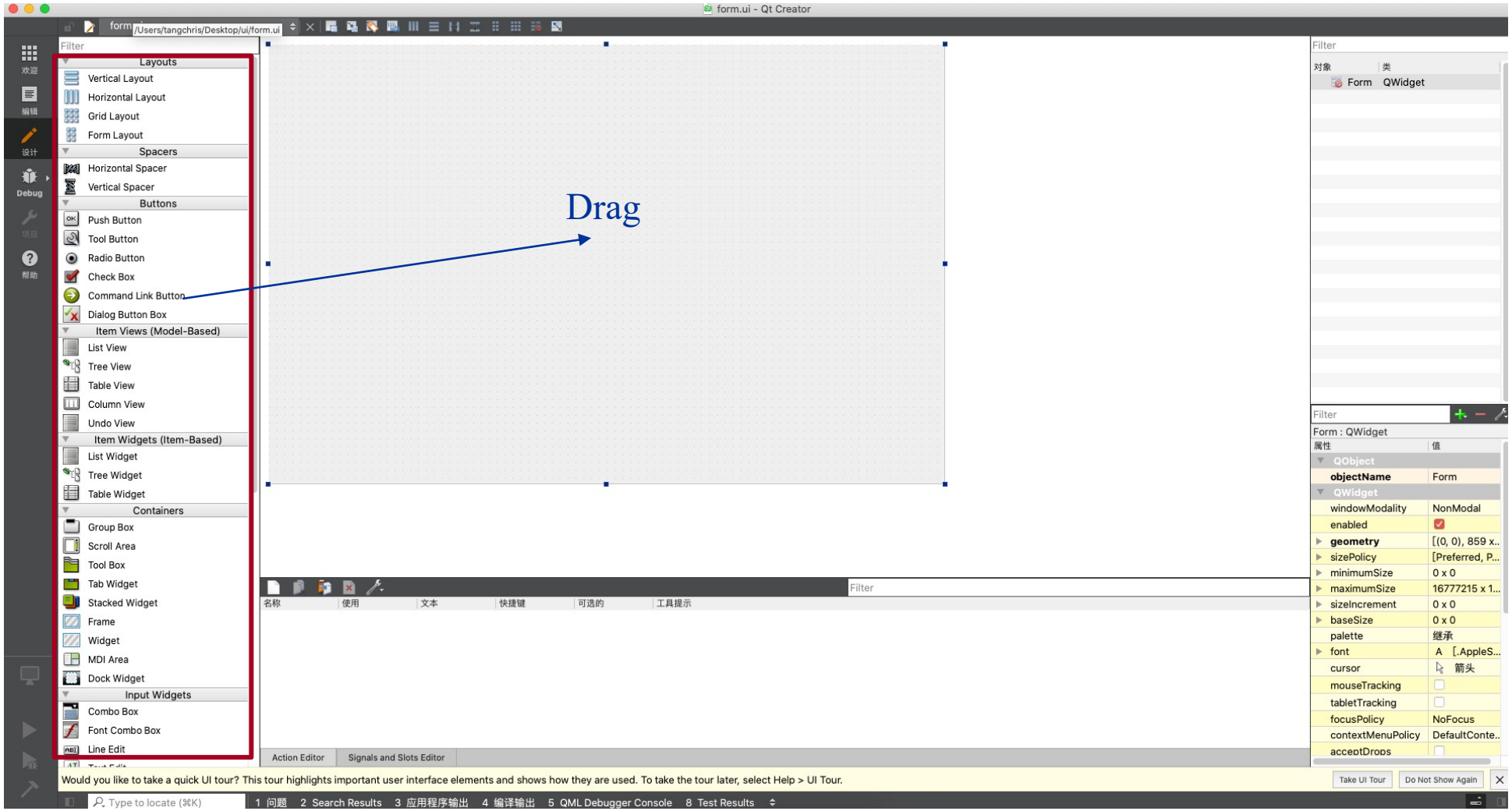


UI Designer (3)

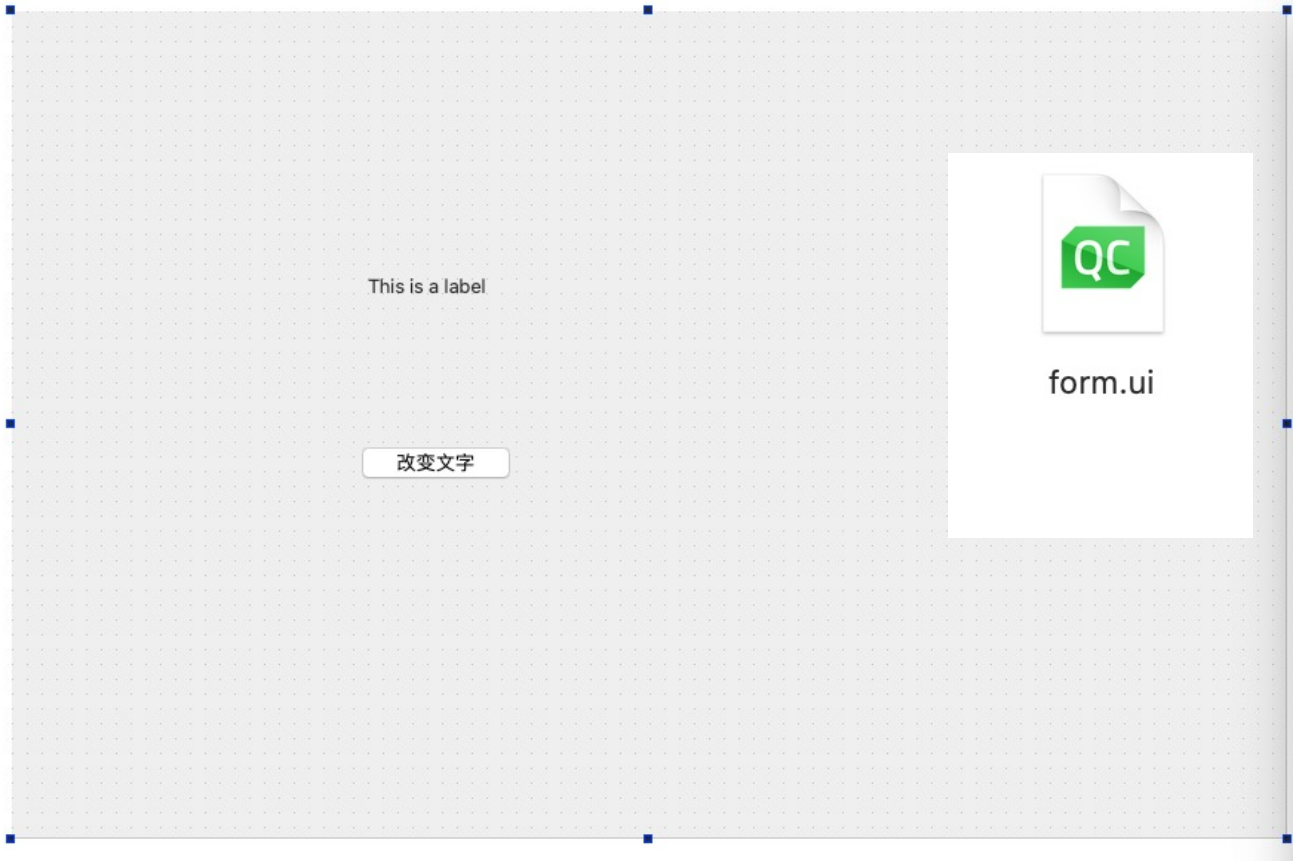
- There are three types of templates: Dialog, Main Window, Widget;
- For demo, we select widget and click continue;



UI Designer (4)



UI Designer (5)




```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>Form</class>
 <widget class="QWidget" name="Form">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>859</width>
    <height>557</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Form</string>
  </property>
  <widget class="QPushButton" name="pushButton">
   <property name="geometry">
    <rect>
     <x>240</x>
     <y>290</y>
     <width>112</width>
     <height>32</height>
    </rect>
   </property>
   <property name="text">
    <string>改变文字</string>
   </property>
  </widget>
  <widget class="QLabel" name="label">
   <property name="geometry">
    <rect>
     <x>240</x>
     <y>170</y>
     <width>141</width>
     <height>31</height>
    </rect>
   </property>
   <property name="text">
    <string>This is a label</string>
   </property>
  </widget>
 </widget>
 <resources/>
 <connections/>
</ui>
```

Loading .ui file in Python

- To load .ui files, we can use the uic module included with PyQt, specifically, the uic.load() method;
- This takes the filename of a UI file and loads it creating a fully-functional PyQT object;

Loading .ui file in Python (2)

```
import sys

from PyQt6 import QtWidgets, uic

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)

    window = uic.loadUi("form.ui")

    window.show()

    app.exec()
```

form.ui

Loading .ui file in Python (3)

- To load a UI from the __init__ block of an existing widget, you can use uic.load (file, self)
-
```
class MainWindow(QtWidgets.QMainWindow):
  def __init__(self):
    super().__init__()
    uic.loadUi("form.ui", self)

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    app.exec()
```

OR Compile .ui into .py file

- Find the pyuic6 (or pyuic6.exe) under the installation folder;
- Add pyuic6 to class PATH;
- Run the following command
- pyuic6 –o <output-python-file-name> <input-ui-file-name>

- Eg. pyuic6 –o form.py form.ui

OR Compile .ui into .py file (2)

```
from PyQt6 import QtCore, QtGui, QtWidgets


class Ui_Form(object):
    def setupUi(self, Form):
        Form.setObjectName("Form")
        Form.resize(859, 557)
        self.pushButton = QtWidgets.QPushButton(Form)
        self.pushButton.setGeometry(QtCore.QRect(230, 290, 112, 32))
        self.pushButton.setCheckable(True)
        self.pushButton.setObjectName("pushButton")
        self.label = QtWidgets.QLabel(Form)
        self.label.setGeometry(QtCore.QRect(240, 170, 141, 31))
        self.label.setObjectName("label")

        self.retranslateUi(Form)
        QtCore.QMetaObject.connectSlotsByName(Form)

    def retranslateUi(self, Form):
        _translate = QtCore.QCoreApplication.translate
        Form.setWindowTitle(_translate("Form", "Form"))
        self.pushButton.setText(_translate("Form", "改变文字"))
        self.label.setText(_translate("Form", "This is a label"))
```

The type of the Form is unbound

OR Compile .ui into .py file (3)

- Import and use the generated py file in project;
-
```
from PyQt6.QtWidgets import QApplication, QWidget
import sys
from form import Ui_Form
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = QWidget()
    uiform = Ui_Form()
    uiform.setupUi(window) // bind the generated UI with window
    window.show()
    app.exec()
```

Signals & Slots

- So far we've created a window and added a simple push button widget to it, but the button doesn't do anything. That's not very useful at all—when you create GUI applications you typically want them to do something!
- What we need is a way to connect the action of pressing the button to making something happen. In Qt, this is provided by signals and slots.

Signals & Slots (2)

- Signals are notifications emitted by widgets when something happens. That something can be any number of things, from pressing a button, to the text of an input box changing, to the text of the window changing. Many signals are initiated by user action, but this is not a rule.

- Slots is the name Qt uses for the receivers of signals. In Python any function (or method) in your application can be used as a slot—simply by connecting the signal to it. If the signal sends data, then the receiving function will receive that data too.

Signals & Slots (3)

- **QPushButton** Signals
- Our simple application currently has a QMainWindow with a QPushButton set as the central widget. Let's start by hooking up this button to a custom Python method. Here we create a simple custom slot named the_button_was_clicked which accepts the clicked signal from the QPushButton.

Signals & Slots (4)

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()  ②

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_clicked)

        # Set the central widget of the Window.
        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec_()
```

Signals & Slots (5)

- You can connect as many slots to a signal as you like and can respond to different versions of signals at the same time on your slots.

```
        button.clicked.connect(self.the_button_was_clicked)
        button.clicked.connect(self.the_button_was_toggled)

        # Set the central widget of the Window.
        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")

    def the_button_was_toggled(self, checked):
        print("Checked?", checked)
```

Widgets

- Widgets API:
- https://www.riverbankcomputing.com/static/Docs/PyQt6/api/qtwidgets/qtwidgets-module.html

QPushButton

- The push button, or command button, is perhaps the most commonly used widget in any graphical user interface. Push (click) a button to command the computer to perform some action, or to answer a question. Typical buttons are OK, Apply, Cancel, Close, Yes, No and Help.

QPushButton (2)

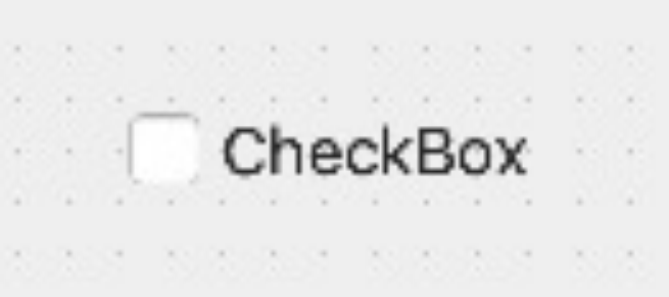
- Signals
  - clicked: This signal is emitted when the button is activated (i.e.,pressed down then released while the mouse cursor is inside the button).
  - pressed: This signal is emitted when the button is pressed down.
  - released: This signal is emitted when the button is released.
  - toggled: This signal is emitted whenever a checkable button changes its state. *checked* is true if the button is checked, or false if the button is unchecked. (Only valid with button.setCheckable(True))

QLabel

- QLabel, arguably one of the simplest widgets available in the QT toolbox.
- widget = QLabel(“Hello”)
- Or, by using the .setText() method to set the text
- widget.setText(“Hello2”)

QCheckBox

- QCheckBox
  - As the name suggests, presents a checkable box to the user. However, as with all QT widgets there are a number of configurable options to change the widget behaviors.
- widget = QCheckBox("This is a checkbox")
- widget.setCheckState(Qt.Checked)
- widget.stateChanged.connect(self.show_state)
- def show_state(): ……



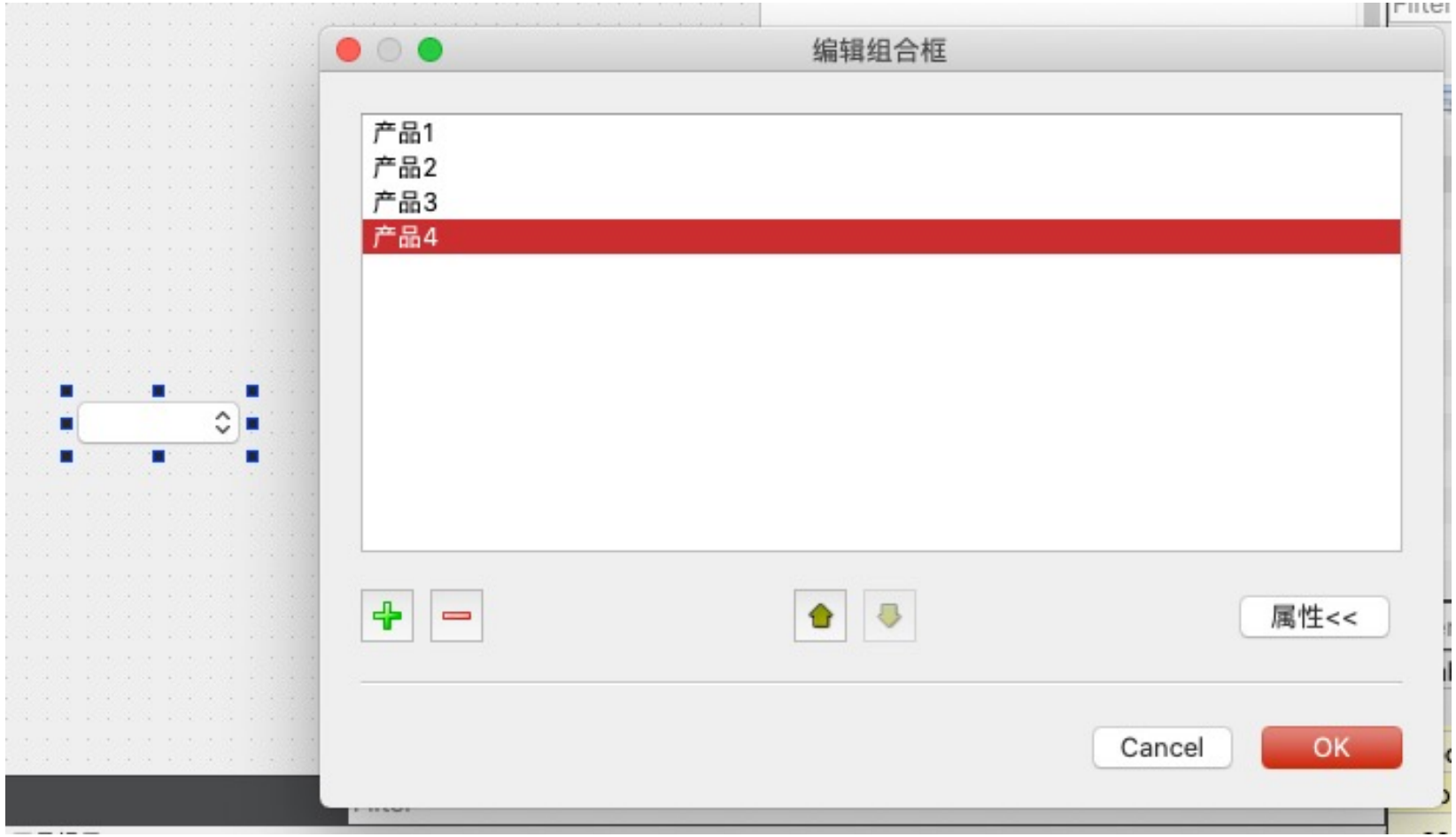
QCheckBox (2)

- You can set a checkbox state programmatically using .setChecked or .setCheckState. The former accepts either True or False representing checked or unchecked respectively.

Flag	Behavior
Qt.Checked	Item is checked
Qt.Unchecked	Item is unchecked

QComboBox

Double Click to edit in the QT Designer



QComboBox (2)

- You can add items to a QComboBox by passing a list of strings to .addItems(). Items will be added in the order they are provided.
- widget = QComboBox()
  widget.addItems(["One", "Two", "Three"])

- widget.currentIndexChanged.connect(self.index_changed)

QComboBox (3)

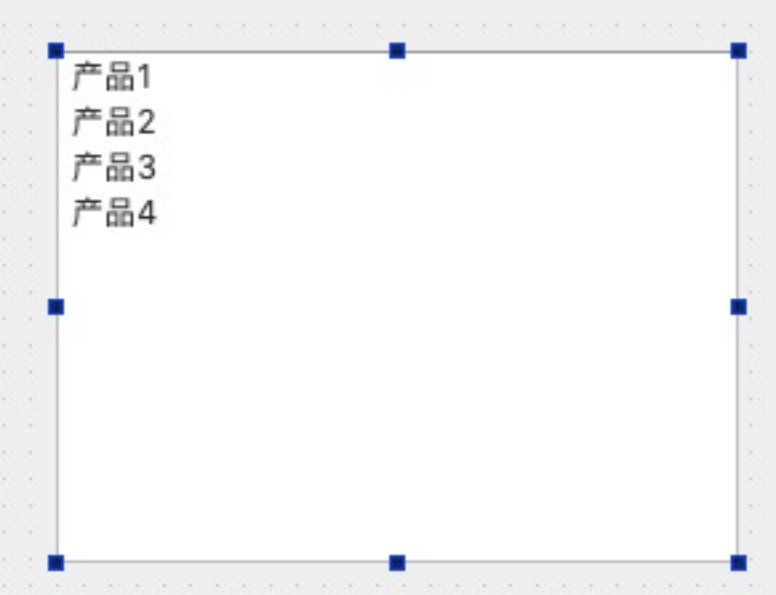
- widget.currentIndexChanged.connect(self.index_changed)
- The .currentIndexChanged signal is triggered when the currently selected item is updated, by default passing the index of the selected item in the list

- def index_changed(self, i): # i is the index

  print(i)

QListWidget

- This widget is similar to QComboBox, except options are presented as a scrollable list of items. It also supports selection of multiple items at once. A QListWidget offers an currentItemChanged signal which sends the QListItem



QListWidget (2)

- widget = QListWidget()
- widget.addItems(["One", "Two", "Three"])
- widget.currentItemChanged.connect(self.index_changed)

- def index_changed(self, i): # Not an index, i is a QListItem
  print(i.text())

QLineEdit

- QLineEdit widget is a simple single-line text editing box, into which user can type input.

- widget = QLineEdit()
- widget.setMaxLength (20)
- widget.setPlaceholderText (“Enter your answer here”)

QLineEdit (2)

- widget.textChanged.connect (self.text_changed)
- widget.textEdited.connect (self.text_edited)

- def text_changed(self, text):
  - print(text)

- def text_edited(self.text)
  - print(text)

QSlider

- QSlider provides a slide-bar widget
- This is often useful when providing adjustment between two extremes, but where absolute accuracy is not required. The most common use of this type of widget is for volume controls
- There is an additional .sliderMoved signal that is triggered whenever the slider moves position

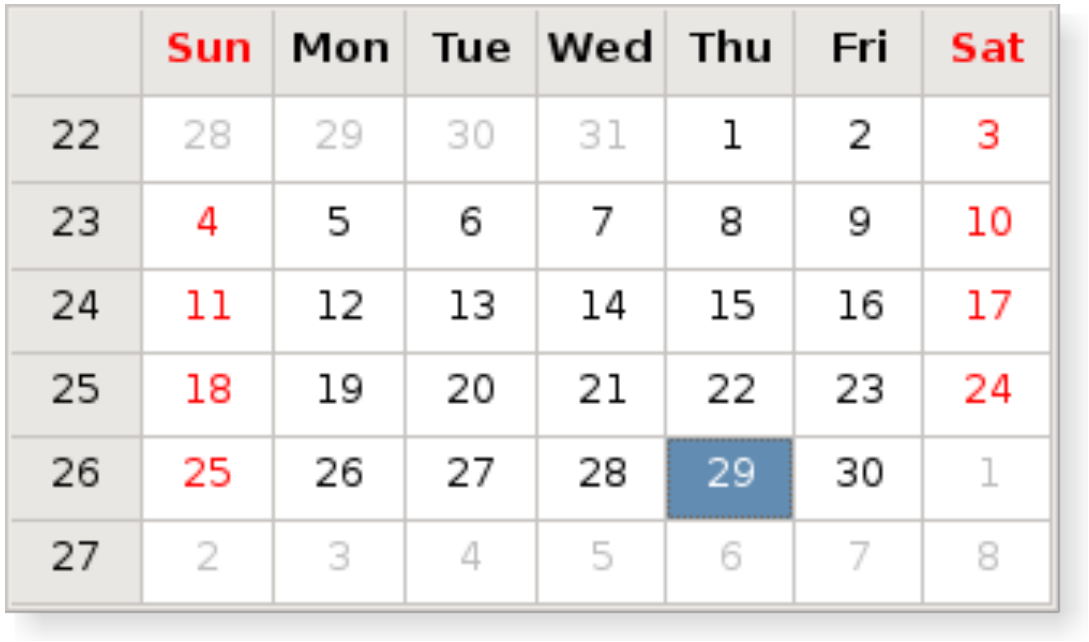
QSlider (2)

- widget = QSider()
- widget.setMinimum (0)
- widget.setMaximum(100)

- widget.setSingleStep(3) //3,6,9,….
- widget.valueChanged.connect(self.value_changed)
- def value_changed(self, i)
  - Print (i)

QCalendarWidget

- QCalendarWidget
- The widget is initialized with the current month and year, but QCalendarWidget provides several public slots to change the year and month that is shown.

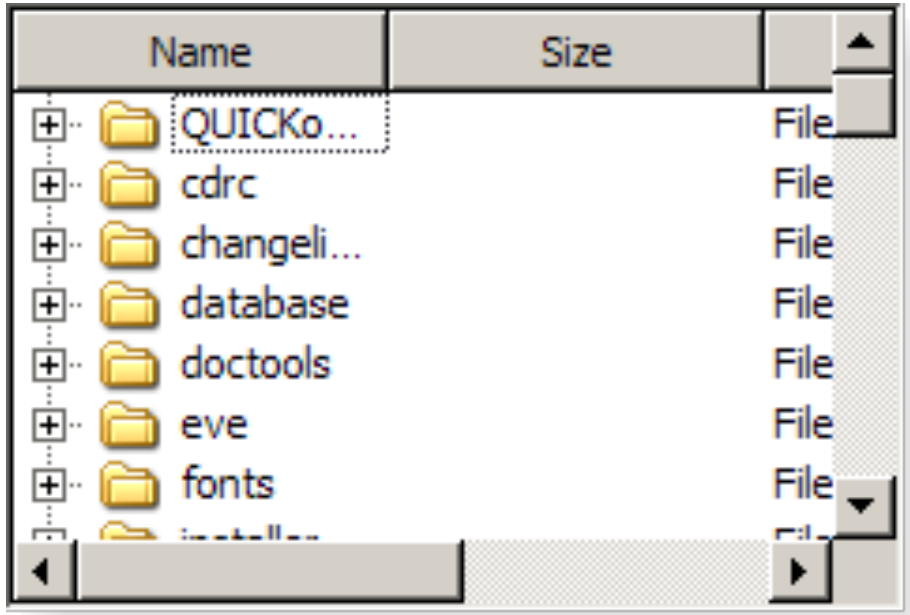


QCalendarWidget (2)

- Signals
  - activated: This signal is emitted whenever the user presses the Return or Enter key or double-clicks a *date* in the calendar widget.
  - clicked: This signal is emitted when a mouse button is clicked. The date the mouse was clicked on is specified by *date*. The signal is only emitted when clicked on a valid date
  - currentPageChanged: This signal is emitted when the currently shown month is changed. The new *year* and *month* are passed as parameters.
  - selectionChanged: This signal is emitted when the currently selected date is changed.

QTreeWidget

- The QTreeWidget class provides a tree view that uses a predefined tree model;
-

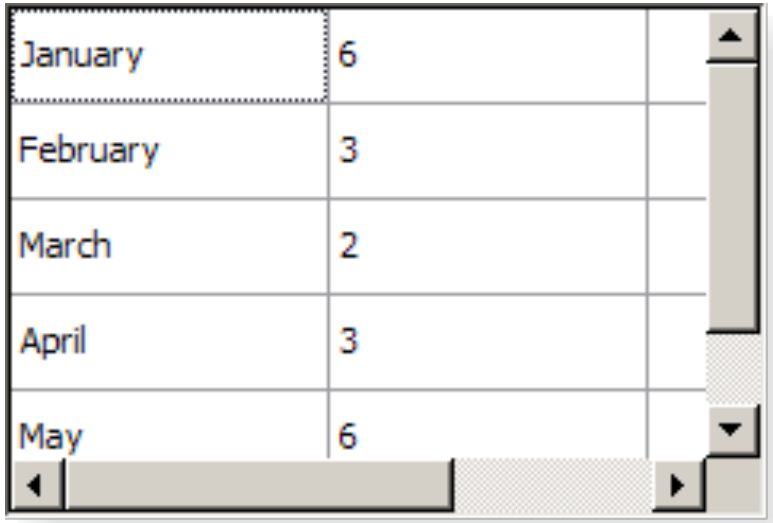


QTreeWidget (2)

- self.tree = QTreeWidget()
- self.tree.setColumnCount(2) # 设置部件的列数为2 self.tree.setHeaderLabels(['Key', 'Value']) # 设置头部信息对应列的标识符
- #设置根节点
- root=QTreeWidgetItem(self.tree)
- root.setText(0,'Root’)
- # #设置子节点1
- child1=QTreeWidgetItem()
- child1.setText(0,'child1')
- child1.setText(1,'ios’)
- root.addChild(child1)

QTableWidget

- The QTableWidget class provides an item-based table view with a default model.
- Table widgets provide standard table display facilities for applications. The items in a QTableWidget are provided by QTableWidgetItem.

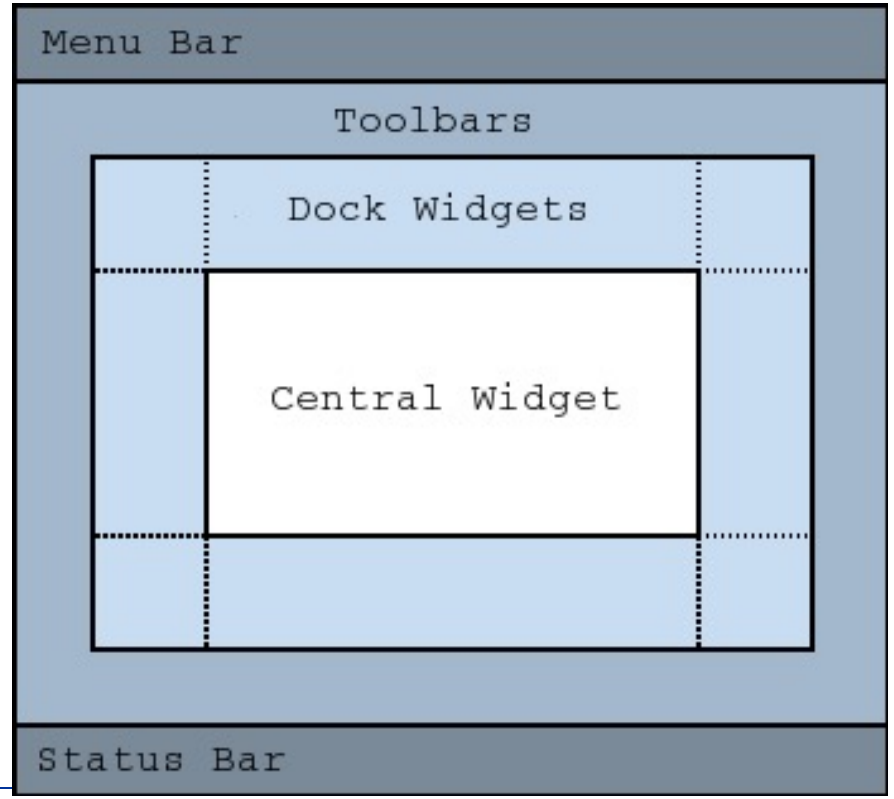


QTableWidget (2)

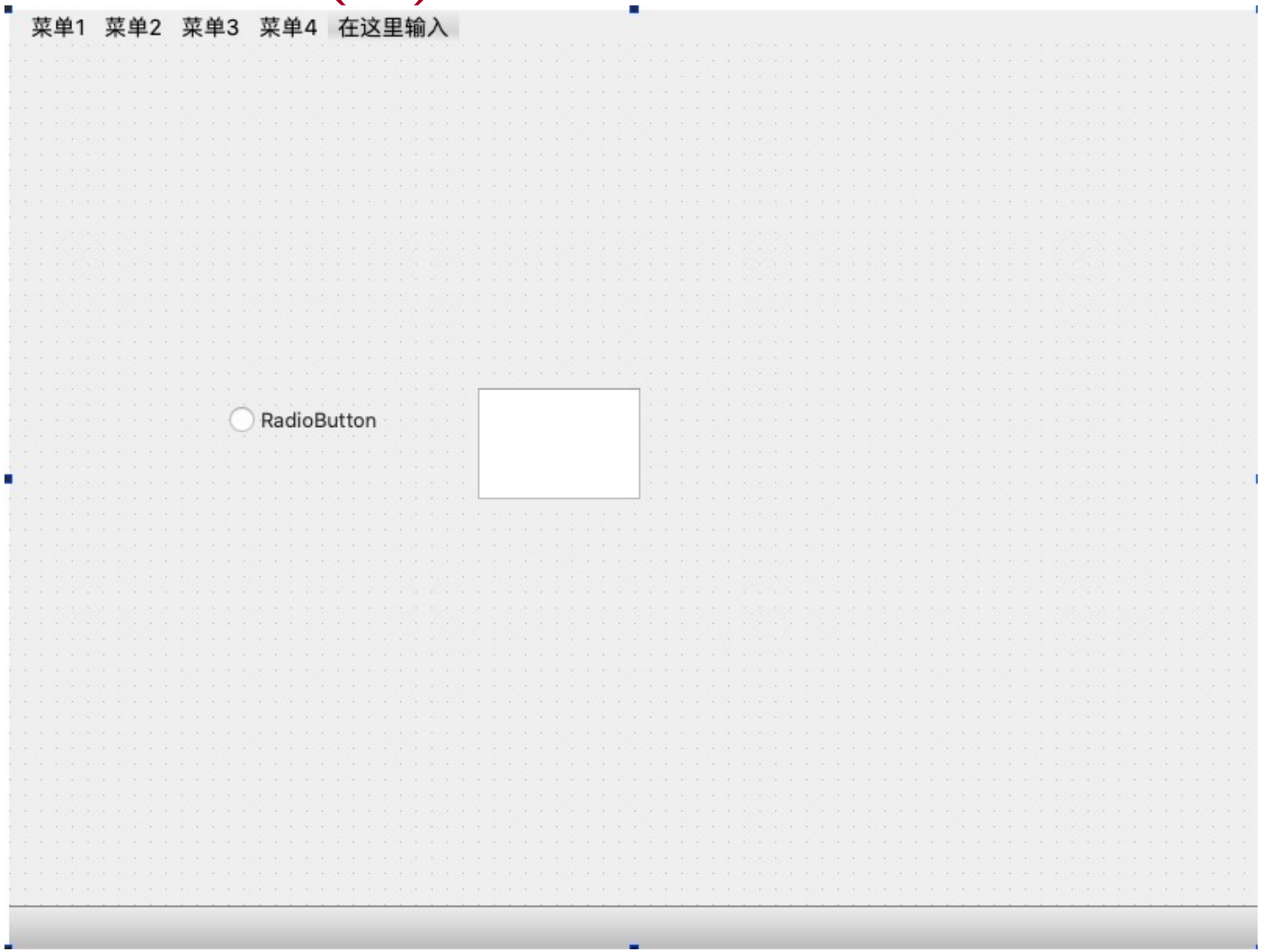
- Signals
  - cellChanged (int, int): This signal is emitted when the cell specified by row and column has changed;
  - cellDoubleClicked(int, int): This signal is emitted whenever a cell in the table is double clicked.
  - currentItentChanged: This signal is emitted whenever the current item changes. The *previous* item is the item that previously had the focus, *current* is the new current item.
  - itemChanged: This signal is emitted whenever the data of *item* has changed.
  - itemClicked: This signal is emitted whenever an item in the table is clicked. The *item* specified is the item that was clicked.

QMainWindow

- A main window provides a framework for building an application's user interface.
- Qt has QMainWindow and its related classes for main window management. QMainWindow has its own layout to which you can add QToolBars, QDockWidgets, a QMenuBar, and a QStatusBar.
- The layout has a center area that can be occupied by any kind of widget. You can see an image of the layout below.



QMainWindow (2)



Reference

- Create GUI Applications with Python & Qt5
- https://paddle.s3.amazonaws.com/fulfillment_downloads/16090/561130/D6eHc7VCSfGrmsulClgK_create-gui-applications-pyqt5.pdf
- Source code of the book http://www.learnpyqt.com/d/pyqt5-source.zip