Building GUIs with Python & PySide

Kris Hardy
ABQpy Meetup
April 7th, 2015
http://github.com/krishardy/abqpy_pyside

What is PySide

- Python wrapper around Nokia's Qt Graphical User Interface library
  - Cross-platform (Windows, Mac, Linux)
    - IOS 8.1 and Android coming in Qt 5
  - Uses native widgets whenever possible
  - http://www.qt.io/developers/
- Released by Nokia
- http://wiki.qt.io/PySideDocumentation
- http://pyside.github.io/docs/pyside/
  - You often will have to go up several layers in the inheritance hierarchy to reach the documentation on core functionality

Similar Frameworks

- PyQt (Qt)
- Tkinter (Tk)
- WxPython (wxwidgets)
- PyGTK (GTK+)
- Pythonqt (for embedding Python in C++ Qt application)

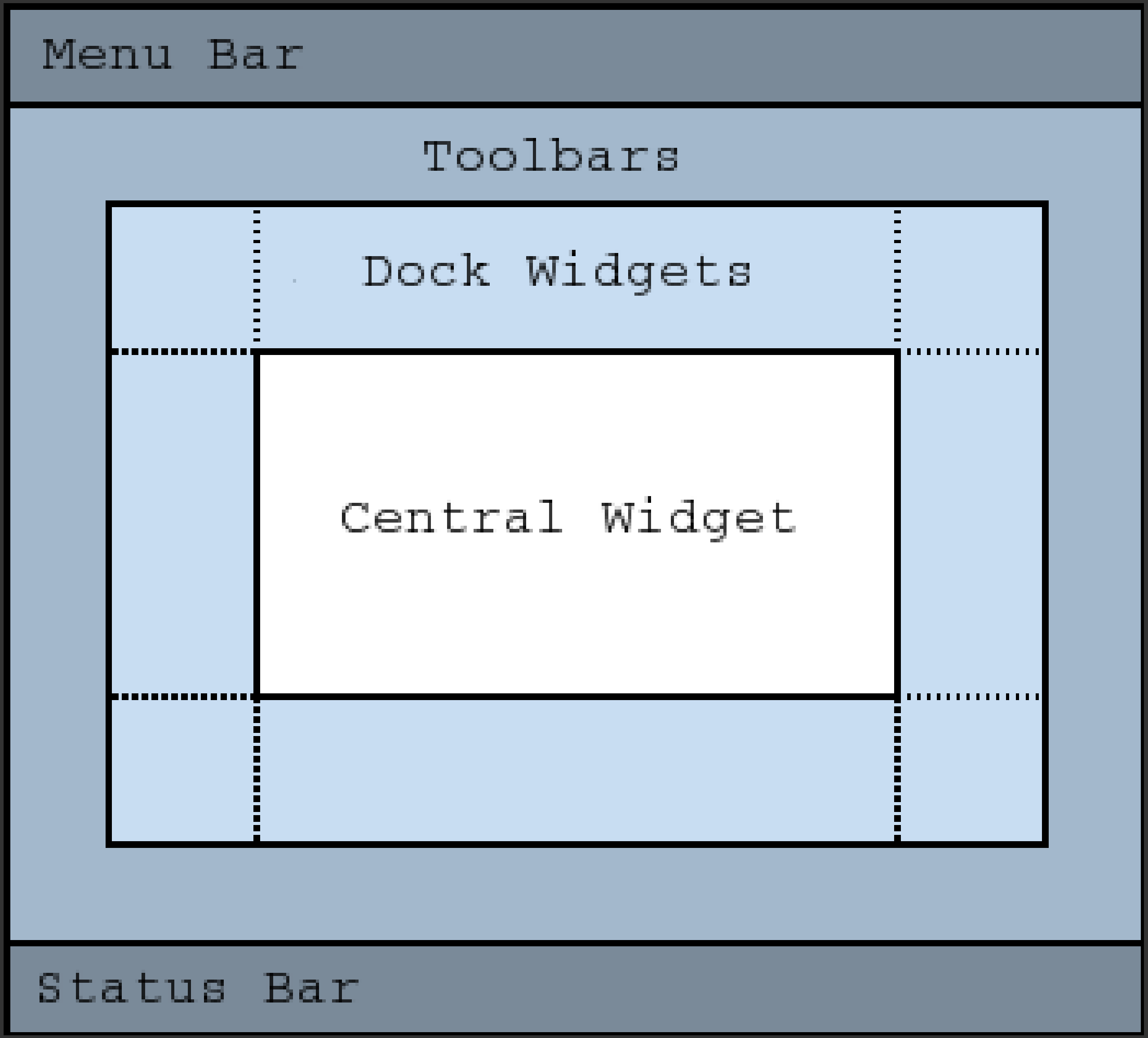
Core Objects

- QObject
  - All Qt objects inherit QObject
  - Provides Signal & Slot handling
- QWidget
  - All UI elements inherit QWidget
  - Provides automatic redrawing
- Signals & Slots
  - Intraprocess communication

1_helloworld

- Simple PySide application
- QApplication.exec_() starts event loop

QMainWindow



2_mainwindow

- QHBoxLayout
  - Automatic horizontal widget layout
- *self.setCentralWidget(centralWidget)*
  - Sets the central widget

Signals & Slots

- Signal
  - Object describing an event (click, value change, etc.)
  - QObject.emit(my_signal): Places my_signal into the UI event queue
  - Signals are emitted by the UI or by your own code
  - Can be custom-built
  - Signals can cause other signals be “emitted”

Signals & **Slots**

- Slots
  - Object method decorated with @QtCore.Slot()
    - Can provide call signatures:
    - @QtCore.Slot(str) # This slot expects a string
    - @QtCore.Slot(str, int) # This one a str and an int
    - ...
  - Can be called by PySide when a signal is emitted
  - Slots can called directly **or** via a signal
    - Calling a slot directly can cause confusion, especially when using QThreads.

3_signalslot

- *@QtCore.Slot()*
  - Decorates a method as a slot
  - Registers the slot with PySide
- *start_button.clicked.connect(self.on_start_click)*
  - Tells PySide to invoke self.on_start_click when a click is emitted by start_button

4_custom_signal

- *updateStatusBar = QtCore.Signal(str)*
  - Creates a custom signal that must be called with a string message
- *@QtCore.Signal(str)*
  - Registers a signal with PySide that expects a string message
- *updateStatusBar.emit(“message”)*
  - Emits the updateStatusBar message, which then causes the connected slot to be called.

5_menu

- Adds menu bar
- Custom and default shortcuts

QMenuBar

- Window menu bar
- *file_menu = self.menuBar().addMenu("&File")*
  - Creates the File menu
- *start_action = file_menu.addAction("&Start")*
  - Creates the Start option in the File menu
- *start_action.setShortcut("Ctrl+S")*
  - Sets the shortcut to <Ctrl>S
- *start_action.triggered.connect(self.on_start_click)*
  - Menu options emit a “triggered” signal when clicked

QKeySequence

- Qt has a bunch of shortcuts that are automaticlly created to match the OS's idiomatic key sequences.
  - Command key on OS X
  - <Ctrl>[vs <Alt><Left>
  - ...
- http://pyside.github.io/docs/pyside/PySide/QtGui/QKeySequence.html#PySide.QtGui.QKeySequence

6_dialog

- QMessageBox
  - Shows modal dialogs for things like info, warnings, confirmations, etc.

Decoupling the UI from heavy work

- Let's set up some code that will do something trivial, but will keep a thread fully busy
- Calculating pi should do it...
- **What will happen when we calculate pi when the Start button is pressed?**

7_pi_nothreads

- UI partially locks up during run (menus and buttons don't respond)
- Possible fixes
  - Set up each iteration of gregory_leibniz on a timer (creates idle time for event handling between iterations)
  - QThreads
  - Python threads
  - Multiprocessing

8_pi_qthreads

- QThreads – Qt-managed threads (I'm not 100% sure of the implementation, but it has bugs and some gottchas)
  - The signals/slots have thread-safe managed queues.
  - If you call a method directly, **YOU** are responsible for managing thread safety
  - I've caused segfaults using QtCore.QThread.currentThreadId() and QtCore.Qthread.currentThread()
    - These are REALLY valuable to ensure that the code really is running in the correct thread, so this is *bad!*
  - GUI elements **MUST** remain in the main thread

QThreads

- Set up and start a bare thread
- Move a worker object to the thread
- Communicate to/from the worker using signals
- You must kill the QThread using my_thread.quit(), then my_thread.wait() to wait for the thread to die

Python Threads

- Seems more reliable than QThreads, but you have no Signal/Slot mechanism.
- You have to set up your own inter-thread communication
  - Queues
  - Events
  - Mutexes
  - ...

9_pi_pythreads

- The Python interpreter keeps all threads inside a single process, on a single core. Good for I/O bound tasks, but not CPU-heavy code.
  - Google: python GIL
- Subclass threading.Thread
- The main loop is the run() method
- I prefer to use Queues for inter-thread communication
- Use blocking calls, but use them wisely

10_pi_multiprocessing

- When doing CPU-intensive work, can run subprocesses on other cores.
- Set up a function to launch as a new process
- I prefer to use multiprocessing.Queue for interprocess communication
- Use blocking calls, but use them wisely
- Crashes can be tricky to debug
  - Hung child processes, dead children, etc.