1.1 Introduction

During a special course, a preproject for my master project, I investigated different options for making graphical user interfaces for python applications. The following sections presents a description and conclusion of this work.

28/3-14, Mads M Pedersen (mmpe@dtu.dk)

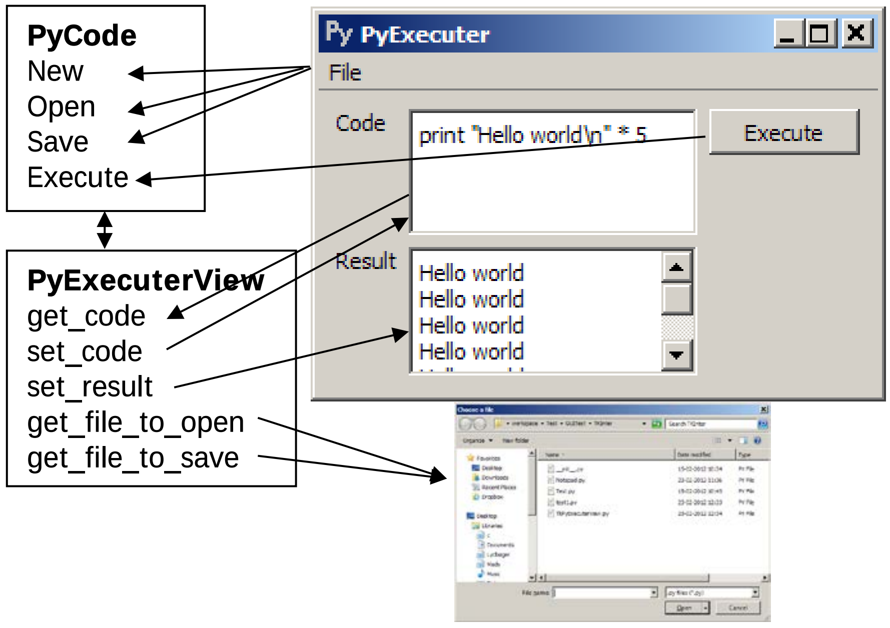
1.2 Graphical User Interface

A graphical user interface, GUI, is required for the post processing tool. The standard Python distribution contains a package, Tkinter, for making GUIs, but more than 30 other GUI options exists.

The GUI package to use must provide up-to-date and good looking visual elements, it should be easy to use and contain all commonly used visual components and dialogs.

In this context four promising GUI packages have been investigated, in order to find out how easy and well a predesigned GUI can be built and how smooth interaction with a model can be implemented. In practise GUIs for a simple Python executer have been built in Tkinter, PyGUI, PyQt and wxPython

The GUIs contain two text fields with corresponding labels, one for code and one for output, an execute button and a menu containing "New", "Open" and "Save". The GUIs instantiate and manipulate a model, `PyCode`, which interacts with the GUI via the interface `PyExecuterView`, i.e. all GUIs inherit from the interface and override its methods, see Figure 10.



```
print "Hello world\n" * 5
```

```
Hello world
Hello world
Hello world
Hello world
```

Figure 1 – GUI design implemented using each of four promising GUI packages

1.2.1. Tkinter

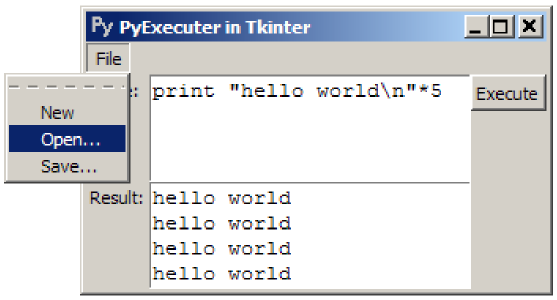
Tkinter is included in the standard Windows install of Python and is considered as Pythons de-facto standard GUI[1]. Tkinter is an interface for a subset of Tk which is an open source, cross-platform widget toolkit providing commonly used widgets and dialogs for desktop applications.

Several visual designers exist for Tkinter. Two, Rapyd-Tk and Visual Python have been tested and found useless – at best they are not finished yet. Therefore code must be written manually. The code is compact and easy to understand, and it is simple to position components in a simple grid, but building a GUI to match the predefined design requires more experience. As seen scrollbars are not added automatically, there is no space between the text fields, there is a dash line in the menu and how to resize the text fields dynamically with the window was not obvious, despite several tutorials and API documentations have been consulted.

```
...
#create labels
Label(root, text="Code:").grid(row=0, sticky=N)
Label(root, text="Result:").grid(row=1, sticky=N)

#Create text fields
self.code = Text(root, height=4, width=25)
self.code.grid(row=0, column=1)
self.result = Text(root, height=4, width=25)
self.result.grid(row=1, column=1))

...
```



```
print "hello world\n"*5
```

```
hello world
hello world
hello world
hello world
```

Figure 2 – Code extract from the initialization method.
Labels and text fields are easily created and positioned

Moreover Tkinter does not support commonly used widgets such as progressbar, combobox, treeview etc.

1.2.2. PyGUI

PyGUI is a cross-platform GUI API that encapsulate different platform API, e.g. win32, Gtk and Cocoa

It is very highlevel in the sense that the standard menus, window title, and shortcuts are automatically created, and partly implemented. When pressing “Open...” for instance, the standard file dialog is shown and the selected file is opened and parsed to a method that must be overridden.

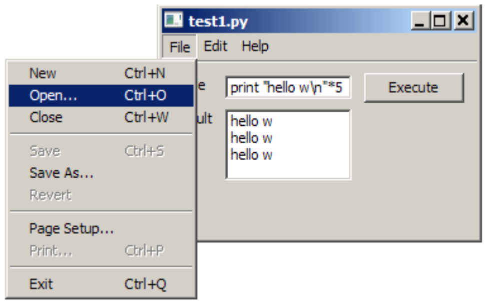
As with Tkinter, the code is compact and easy to read, and components are easily position in a grid, but according to the only reference[2] I have found, text fields have no height attributes and the grid layout allows only one row to be expanded. As seen in Figure 12, the result is not impressing as only one text field can have more than one line, and the width of the fields cannot be adjusted either.

[1] http://wiki.python.org/moin/TkInter

[2] http://www.cosc.canterbury.ac.nz/greg.ewing/python_gui/version/Doc/index.html

```
...
self.code = TextField(multiline=True)
self.result = TextField(multiline=True)
self.btnExecute = Button("Execute")
self.lblCode = Label("Code")
self.lblResult = Label("Result")
g = [[self.lblCode, self.code, self.btnExecute],
     [self.lblResult, self.result, None]]
self.grid = Grid(g, padding=(10, 10), expand_row=1)

...
```



```
print "hello w\n"*5
```

```
hello w
hello w
hello w
```

Figure 3 – PyGui creates standard menus and shortcuts automatically

Unfortunately PyGUI is also limited to the most common standard components, it is not obvious what is done automatically or how to do things, and the API reference is ok, but very short.

1.2.3. Qt

Qt is a C++ cross-platform application and user-interface framework. It is developed by Nokia and distributed under both open source and commercial licenses.

Qt contains a comprehensive library of GUI classes, but also modules with features for network support, database access, XML parsing, OpenGL 3D rendering, SVG graphic displaying and low level multimedia features.

The Qt framework can be integrated in Python programs via bindings provided by PyQT or PySide. Both libraries allow the programmer to create GUIs via Qt Designer. Qt Designer is a visual editor that allows the user to build a GUI of Qt widgets using drag and drop, see Figure 13. The result is a .ui file that can be imported and used directly in Python programs or it can be compiled to a python class. Finally GUIs can also be written manually by the programmer.

Both binding libraries are available for Python 2.7 and 3.2.

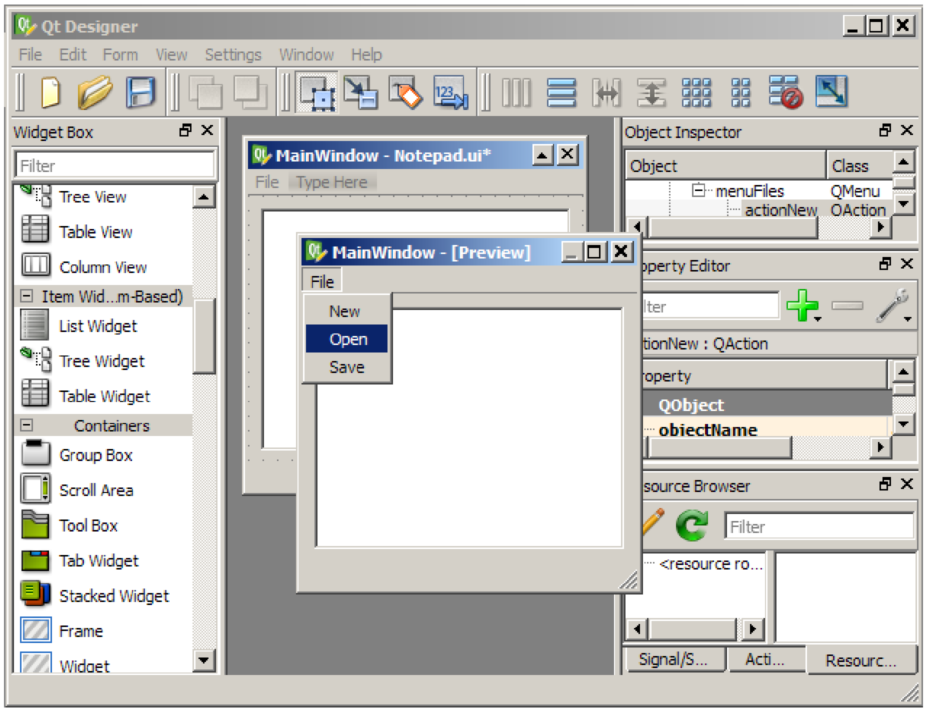


Figure 4 – Qt designer – a visual editor for Qt widgets

With a little experience it is easy to build GUIs exactly as intended and it seems as there is almost no limitations.

Due to multiple nested layouts and auto generated and compiled code, the generated Python code is long and obscure, but in this case it does not matter as it can be used as a black box.

In Tkinter and PyGui a function object was passed to the button and pressing the button invoked the function. In Qt, widget events, e.g. click, hover, resize etc, can be bound to a slot in a receiver, e.g. `Execute()` in `MainWindow`. These bindings are made in Qt designer as well.

Since the auto generated Python file is awkward and in addition overwritten every time a modification is saved and compiled, it is inconvenient to implement slots in this file. Instead the design outlined in Figure 14 can be used. In this design the top class, which inherits from `QMainWindow`, sets the black box GUI as its userinterface, `ui`, and since it is now `MainWindow`, the former mentioned `Execute()` slot can be implemented here.

```python
from GUITest.PyQT.qt_exe_view import QtPyExeViewGUI

class QtPyExeView(QtGui.QMainWindow, PyExecuterView):

    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.ui = QtPyExeViewGUI.Ui_MainWindow()
        self.ui.setupUi(self)
        #or self.ui = uic.loadUi("QtPyExeViewGUI.ui", self)

    def Execute(self):
        self.pyCode.execute()

...
```

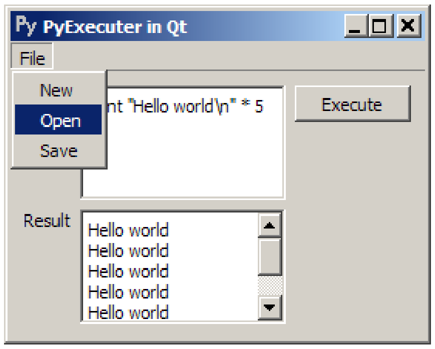


Figure 5 – The Qt GUI is build in Qt designer, compiled to a Python class and used as a black box

1.2.4. wxPython

wxPython is a Python wrapper for wxWidgets, a cross-platform GUI API. It contains a rich toolbox of widgets, including datepicker, treeview gridview etc.

It is possible to build the GUI by manually coding or to design it in the visual editor wxGlade. wxGlade is less fancy than Qt Designer and during test it did not repaint correct, some bugs were revealed, some changes were not visible in the design window, but required a "preview", and I did not succeed making text fields that kept their width in preview, see in Figure 15. However it is relatively easy to use and created GUIs are converted to Python classes by a single click. The API is documented by an auto generated reference, but it lacks descriptions, explanations and examples.

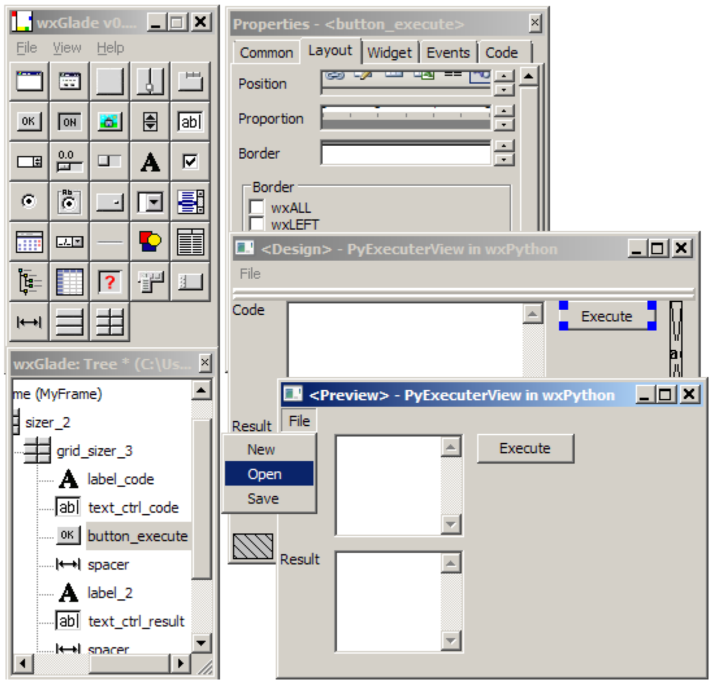


Figure 6 - wxGlade a visual designer for wxPython

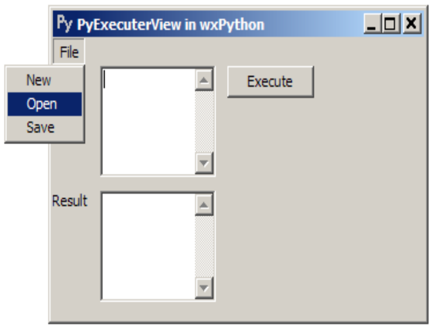
It is not obvious how event handlers can be implemented without editing the auto generated file, but by the use of comment markers the auto generator should be able to replace its own code only, but it seems a little risky to rely on.

```
  ...
  self.label_code = wx.StaticText(self, -1, "Code")
  self.text_c= wx.TextCtrl(self, -1,"",style=wx.TE_MULTILINE)
  self.button_execute = wx.Button(self, -1, "Execute")
  self.label_2 = wx.StaticText(self, -1, "Result")
  self.text_r = wx.TextCtrl(self,-1,"",style=wx.TE_MULTILINE)
  ...
  self.Bind(wx.EVT_BUTTON, self.Execute, self.button_execute)


def __do_layout(self):
  # begin wxGlade: MyFrame.__do_layout
  sizer_2 = wx.BoxSizer(wx.VERTICAL)
  grid_sizer_3 = wx.FlexGridSizer(3, 4, 10, 10)
  grid_sizer_3.Add(self.label_code, 0, 0, 0)
  grid_sizer_3.Add(self.text_ctrl_code, 1, wx.EXPAND, 0)
  grid_sizer_3.Add(self.button_execute, 0, 0, 0)
  ...
  # end wxGlade

 def Execute(self, event): # wxGlade: MyFrame.<event_handler>
  self.pyCode.execute()

 ...
```



1.2.5. Conclusion

As PyQt using Qt Designer provides the most comprehensive widget collection, is easiest to use and allows the user to build GUIs that appears as intended, this package will be used.