POWERING POTENTIAL

Python 3 Programming

Appendix
Introduction to Tk

INTRODUCTION TO TK

Graphical User Interfaces
Overview of Python Tk
Requirements for Python Tk
- Named arguments
- Subroutine references
- Closures

Tk design
- Event-driven
- Widget hierarchy
- Dynamic widget size & position

A brief introduction to the tkinter modules.

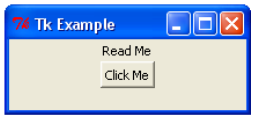
Graphical user interfaces from Python

- **Python supports various GUI extensions**
  - X11
  - Win32
  - Macintosh
  - Gtk (X specific)
  - Tk
  - Qt
- **Tk is most commonly used**
  - Tkinter on Python 2, tkinter on Python 3
  - Portable across UNIX and Windows
  - Based on tcl/Tk toolkit
  - Object-oriented interface

Python supports a number of GUI-extensions, most of which are tied to a specific windowing system (such as X11, Win32 or Macintosh). The Gtk extension is from the Gimp Toolkit, which also runs on Windows.
The Tk extension is most commonly used, because it is portable across platforms, powerful and easy to use. It is based on the Tk toolkit from the tcl language, developed by John Ousterhout (first at Berkeley, then at Sun and now at Scriptics). The same Tk toolkit has also been ported to Perl, Scheme and Guile.
The main difference between Python Tk and tcl/Tk is that the perl version is fully object-oriented, whereas the tcl version is string based (and generally a mess).

A simple example

- **Create objects**
  - Main window
  - Labels and buttons
  - Define call back
  - Invoke main loop




```
from tkinter import *

def button_proc():
    print('Call-back function for button')

root = Tk()
root.title('Tk Example')
Label(root, text='Read Me').pack()
Button(root, text='Click Me',
       command=button_proc).pack()
root.mainloop()
```

elements.py

The example above starts by including the Tk module; this is required for every perl/Tk program. It then creates a new MainWindow object. Evidently, Tk plays a few tricks here: the Tk module has defined a MainWindow class as well.
Given a main window, a label and a button are defined. By creating these from the main window, Tk knows how the widgets have to be nested. Evident is that Tk uses a named-argument calling convention. The **`pack()`** calls are necessary to handle widget layout and are described in more detail later.
Finally, the main loop is invoked. Tk now takes over and handles all keyboard, mouse and window events. Whenever something noteworthy happens, user-defined call-back routines are invoked that handle an event. This is an example of event-driven programming.

Elements of Python Tkinter

- **Event-driven**
  - Create objects, then run a main loop
  - Main loop (indirectly) invokes callback subroutines
- **Methods use named arguments**
  - Default values exist for most arguments
- **Events invoke callback subroutines**
  - Reference to subroutine
  - Anonymous subroutine
  - Closure
  - Object + method name

Tk is like most other graphical environments in that you have to give up control. ("Don't call us – we'll call you").
So, instead of tracking all events and deciding what action to take, you set up the graphical environment and start the main loop, which will then handle all default actions and invoke your code when necessary to take an action.

User input is hidden

- **Data entry objects handle input**
  - Stored in a StringVar, IntVar variable
  - No need to query state


```
from tkinter import *

def button_proc():
    print('Call-back function, printer is now', ent.get())

root = Tk()
root.title('Printer Options')

but = Button(root, text='Done', command=button_proc)
but.pack(side = 'bottom')
Label(text='Printer Name').pack(side='left')

printer = StringVar()
ent=Entry(root, textvariable=printer)
ent.pack(side='left')

printer.set('SAP64')
root.mainloop()
```

printer.py

One of the nicest features of Tk is that, unlike many other GUI toolkits, there is no need to query the user interface elements for the state of options. Instead, communication is done through a reference – the GUI sets a variable which can be read in the call-back subroutines.
This communication is two-way: if a call-back routine modifies the variable, the GUI will change its display accordingly.

Widgets are hierarchical

Widgets inside widgets inside...

- Normally indicated through order of creation and choice of parent

```
from tkinter import *

root = Tk()

topf = Frame(root).pack()
botf = Frame(root).pack()

Label(topf, text ='One').pack(side='left')
Label(topf, text ='Two').pack(side='left')

Label(botf, text='Three').pack(side='left')
Label(botf, text='Four').pack(side='left')

root.mainloop()
```

hierarchy.py

In Tk, choosing your parent wisely is even more important than in real life! In the example above, the main window contains two Frames (invisible place holders) which in turn contain buttons. This kind of nesting is typical for Tk – instead of using a fixed widget setup with absolute positioning, the nesting order creates a flexible and dynamic layout.
This idea is also used in Java, where the AWT uses both nested widgets and so-called layout managers to determine the relative layout of the user interface.

Widgets are dynamic

Widgets may be added or removed while running

- All widgets dynamically sized

```
from tkinter import *

root = Tk()
button = Button(root)
button.pack()
label  = Label(root, text="I'm here")

def hide_label():
    label.forget();
    button.configure(text="Show Label",
                     command=show_label)

def show_label():
    label.pack()
    button.configure(text="Hide Label",
                     command=hide_label)

show_label()
root.mainloop()
```

Initial size is computed and resized when window size changes

dynamic.py

The example above is slightly unusual in that we hold on to the widgets using global variables and use those inside the call-back routines. As we'll see in the next chapters, you can use either closures or named widgets and dynamic enquiry functions to achieve the same in a neater fashion.
The code above creates the Label widget once and dynamically displays or hides it, using pack and packForget. The Button widget is always there, but is re-configured dynamically to change its text and callback functions.

Building a main window

- A main window also requires title, icon, and menus

```
from tkinter import *
root = Tk()
root.title('Window Title')
root.wm_iconbitmap('qa.ico')

mbar = Menu(root)
filemenu = Menu(mbar, tearoff=0)
filemenu.add_command(label='Quit',
                     command=lambda: root.destroy())
helpmenu = Menu(mbar, tearoff=0)
helpmenu.add_command(label='RTFM')

mbar.add_cascade(label='File', menu=filemenu)
mbar.add_cascade(label='Help', menu=helpmenu)
root.config(menu=mbar)

bt = Button(root, text ='Click Me!', command=button_proc)
bt.pack()

root.mainloop()
```

mainwindow.py

A main window should always have a title and an icon. The icon bitmap must either be one of the pre-defined bitmaps, or can start with an @ to indicate a filename of an xpm file.
In addition, most main windows should have a menu with a ‘File – Quit’ entry.

SUMMARY

- **Tk is portable**
- Normally bundled with the base
- **User input is hidden**
- **Widgets are hierarchical**
- **Widgets are dynamic**
- **Start with a main window**