

Tkinter Geometry Managers

Note: *this is a work in progress*

Introduction

As we've seen, creating a new widget doesn't mean that it appears on the screen. To display it, you need to call a special method: either **grid**, **pack** (which we've used in all examples this far), or **place**.

The **pack** method doesn't really display the widget; it adds the widget to a list of widgets managed by the parent widget. An idle task [TERM?] will take care of the rest as soon as Tkinter has nothing better to do.

Here's a simple example:

```
from Tkinter import *

root = Tk()

label = Label(root, text="Message")
label.pack()

button = Button(root, text="Quit", command=root.destroy)
button.pack()

mainloop()
```

The first call to **pack** simply adds the **label** widget to the list of widgets managed by the **root** window (which behaves pretty much like any other **Toplevel** widget). It also schedules an idle task [TERM?] which will call the geometry manager at a later time.

The second call adds the **button** widget to the **root** list. The idle task has still not run, so this method doesn't add a new one.

Finally, the **mainloop** will sooner or later get around to call the idle task. The geometry manager then loops over all its children in one go, calculates the appropriate size for each of them based on their "natural" size [TERM?], displays them, and finally asks the root window to resize itself.

Note that the use of an idle task means that the geometry manager isn't directly involved by the call to **pack**. This allows you to pack a whole bunch of widgets, and have them all managed in a single pass by the manager's geometry algorithms.

Now, let's take a closer look at the different geometry managers provided by Tkinter.

The Grid Manager

As the sound implies, this manager organizes its children in a grid. The master widget is split into a number of rows and columns, and each "cell" in this grid can hold a single widget.

[image goes here]

To use the grid manager for a widget, use the **grid** method:

grid(row=index, column=index, options...). This method places the widget in the given cell. The grid manager will make sure that both the row and the column is large enough to display the entire widget, and position it correctly.

If you leave out the **row** option, the widget is placed in the first empty row. If you leave out the **column** option, the widget is placed in the leftmost column (index 0).

```
# Example: File: grid-example-1.py
```

```

from Tkinter import *

root = Tk()

w = Label(root, text="Additive:")
w.grid(sticky=E)
w = Label(root, text="Subtractive:")
w.grid(sticky=E)

w = Label(root, text="Cyan", bg="cyan", height=2)
w.grid(row=1, column=1)
w = Label(root, text="Magenta", bg="magenta", fg="white")
w.grid(row=1, column=2)
w = Label(root, text="Yellow", bg="yellow", height=2)
w.grid(row=1, column=3)

w = Label(root, text="Red", bg="red", fg="white", height=2)
w.grid(row=0, column=1)
w = Label(root, text="Green", bg="green", height=3)
w.grid(row=0, column=2)
w = Label(root, text="Blue", bg="blue", fg="white")
w.grid(row=0, column=3)

mainloop()

```

Things to Notice

Rows and columns are made large enough to fit the largest widget. The labels have different widths, and I've used the **height** option to vary their height. Note how each row and column are made wide enough to show all widgets, and that the widgets are centered in the resulting cells. You can use options to change this in different ways; see the next section for details.

Using default values for row and column. The first two labels use default values for the **row** and **column** options. As a result, they are both placed in column 0, in the rows 0 and 1.

You can place the widgets into the grid in any order. It's only the row and column numbers that count when the grid manager lays the widgets out. Although it's not shown here, you can also leave empty rows and columns if that makes your code simpler.

Rows, Columns, and Cells

[minsize] [pad]

The **grid_size** method returns the number of columns and rows allocated by the grid manager. You can use this function to add new rows (or columns) to a grid:

```

def add_entry(master, text):

    column, row = master.grid_size()

    label = Label(master, text=text)
    label.grid(row=row, column=0, sticky=E, padx=2)

    entry = Entry(master)
    entry.grid(row=row, column=1, sticky=E+W)

    return entry

```

[grid_location]

Expanding and Filling

By default, the grid manager lets each widget keep its natural size [TERM?]. If the cell turns out to be larger than this size, the widget is placed in the middle of the cell.

You can use the **sticky** option to modify this behavior. This option tells the geometry manager to “attach” the widget to one or more of the cell borders.

[ILLUSTRATION]

The option value is a string, which can contain one or more of the characters “n” (north), “s” (south), “w” (west), and “e” (east). For example, “n” centers the widget along the upper cell border. “se” moves the widget to the lower right corner.

If you attach the widget to two opposite borders, it’s resized as necessary. For example, “ns” stretches the widget vertically, while “we” stretches it horizontally. Finally, “nwse” makes the widget fill the entire cell.

Instead of string values, you can use the builtin constants N, S, E, W, NE, NW, SE, and SW. To combine these, use the “+” operator.

[grid_columnconfigure grid_rowconfigure weight]

[padding]

Spans

Usually, widgets occupy a single cell in the grid. You can use the **columnspan** and **rowspan** options to change this for any given widget.

The **columnspan** option tells the geometry manager that a widget should occupy not only its original cell, but also one or more extra cells to the right. The option value is the total number of cells to occupy.

The **rowspan** option is similar, but it tells the manager to occupy extra cells under the original.

[example]

Geometry Propagation

[grid_propagate]

[example]

[note: continues in cho9b.txt]

The Pack Manager

The pack manager places its children on top of each other in columns, or side by side in rows.

This manager is a quite elaborate piece of code. Too elaborate to fully understand in one sitting, perhaps, so I’m not going to tell you the full story until later. Let’s start with a simplified version of the story.

“The Pack Geometry Manager For Dummies”

This far, we’ve learned two things:

1. To make a widget visible, use a geometry manager
2. The grid geometry manager is a geometry manager

Sounds like all we need to know, doesn’t it? Why learn another geometry manager?

Well, there is a reason why I used the pack manager in all examples leading up to this chapter. The pack manager is much easier to use in a few, but quite common situations:

1. Put a widget inside a frame (or any other container widget), and have it fill the entire frame
2. Place a number of widgets **on top of each other**

3. Place a number of widgets **side by side**

Filling a Frame (Or Any Other Container Widget)

A common situation is when you want to place a widget inside a container widget, and have it fill the entire parent. Here's a simple example: a listbox placed in the root window:

```
# Example: File: pack-listbox-1.py

from Tkinter import *

root = Tk()

listbox = Listbox(root)
listbox.pack()

for i in range(20):
    listbox.insert(END, str(i))

mainloop()
```

By default, the listbox is made large enough to show 10 items. But this listbox contains twice as many. Wouldn't it be reasonable if we could show them all simply by increasing the size the root window?

Of course, it isn't that easy. The listbox [FIXME]

Packing Things On Top Of Each Other

To put a number of widgets in a column, use the **pack** method without any options:

pack(). This method centers the widget along the top border of the parent widget. If other widgets are packed in the same parent, this widget is placed just under them.

```
# Example: File: pack-example-1.py

from Tkinter import *

root = Tk()

w = Label(root, text="Red", bg="red", fg="white")
w.pack()
w = Label(root, text="Green", bg="green", fg="black")
w.pack()
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack()

mainloop()
```

You can use the **fill=X** option to make all widgets as wide as the parent widget:

```
# Example: File: pack-example-2.py

from Tkinter import *

root = Tk()

w = Label(root, text="Red", bg="red", fg="white")
w.pack(fill=X)
w = Label(root, text="Green", bg="green", fg="black")
w.pack(fill=X)
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack(fill=X)

mainloop()
```

Packing Widgets Side By Side

To pack widgets side by side, use the **side** option. If you wish to make the widgets as high as the parent, use the **fill=Y** option too:

```
# Example: File: pack-example-3.py

from Tkinter import *
```

```

root = Tk()

w = Label(root, text="Red", bg="red", fg="white")
w.pack(side=LEFT)
w = Label(root, text="Green", bg="green", fg="black")
w.pack(side=LEFT)
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack(side=LEFT)

mainloop()

```

If you need to create more complex layouts, the standard answer is to use the grid manager instead, or to use nested **Frame** widgets. For example, here's a simple entry form using frames within frames:

```

# Example: File: pack-entry-form-1.py

from Tkinter import *

root = Tk()

def add_entry(master, text):

    frame = Frame(master)

    label = Label(frame, text=text)
    label.pack(side=LEFT)

    entry = Entry(frame)
    entry.pack(side=LEFT)

    frame.pack()

add_entry(root, "First")
add_entry(root, "Second")
add_entry(root, "Third")

mainloop()

```

It might be simple, but it's no simpler than the corresponding grid code, and the result is much worse. The main problem is that since the labels have different widths, the entry fields don't line up. You could use the **width** option to make all the labels equally wide:

```

# Example: From file: pack-entry-form-2.py

def add_entry(master, text):

    frame = Frame(master)

    label = Label(frame, text=text, width=10)
    label.pack(side=LEFT)

    entry = Entry(frame)
    entry.pack(side=LEFT)

    frame.pack()

```

This has some drawbacks, including the problem of picking an appropriate width. If width is too small, the **Label** widget will not display the entire label.

A less obvious way is to tweak the packer options a little. First, use the **fill** option to make all frames as wide as the parent widget, and then pack the entry field towards the **right** border:

```

# Example: From file: pack-entry-form-3.py (portions)

def add_entry(master, text):

    frame = Frame(master)

    label = Label(frame, text=text)
    label.pack(side=LEFT)

    entry = Entry(frame)
    entry.pack(side=RIGHT)

```

```
frame.pack(fill=X)
```

Much better, but the labels are still left justified. What if we want them to be right justified, like in the grid examples?

Well, nothing is impossible. Just pack the labels **after** the entry widget, and pack them against the right border. This way, they will end up just to the left of the entry fields:

```
def add_entry(master, text):  
  
    frame = Frame(master)  
  
    entry = Entry(frame)  
    entry.pack(side=RIGHT)  
  
    label = Label(frame, text=text)  
    label.pack(side=RIGHT)  
  
    frame.pack(fill=X)
```

That's more like it. But given that this is not very obvious, and no shorter than the corresponding grid code, it's probably a good idea to use the grid manager for cases like this.

How Things Really Work

[FIXME]

By default, the areas used by the pack manager is similar to the cells used by the grid managers. If at all possible, the area is made large enough to hold the widget,

However, some of the details are a bit less obvious in this case. If you pack a widget along the top or bottom side, the area allocated to that widget is high enough to display the entire widget, but as wide as the **entire** parent widget.

Likewise, if you pack a widget along the left or right side, the area is as wide as the widget, but as high as the **entire** parent widget.

FIXME lets each widget keep its natural size [TERM?]. If the cell turns out to be larger than this size, the widget is placed in the middle of the cell.

You can use the **sticky** option to modify this behavior. This option tells the geometry manager to “attach” the widget to one or more of the cell borders.

Geometry Propagation

The Place Manager


FIXME

The Notebook Component

FIXME

Rolling Your Own Geometry Manager

FIXME

 rendered by a [django](#) application, hosted by [webfaction](#).