Essentials

Tutorial

Show: All Languages ▼

This tutorial will quickly get you up and running with the latest Tk from Tcl, Ruby, Perl or Python on Mac, Windows or Linux. It provides all the essentials about core Tk concepts, the various widgets, layout, events and more that you need for your application.

Previous: Basic Widgets (widgets.html) | Contents (index.html) | Single Page (onepage.html) | Next: More Widgets (morewidgets.html)

The Grid Geometry Manager

We'll take a bit of a break from talking about different widgets (what to put onscreen), and focus instead on geometry management (where to put it). We introduced the general idea of geometry management in the "Tk Concepts" chapter; here, we focus on one specific geometry manager: grid.

As you've seen, grid lets you layout widgets in columns and rows. If you're familiar with using HTML tables to do layout, you'll feel right at home here. This chapter describes the various ways you can tweak grid to give you all the control you need for your user interface.

Grid is one of several geometry managers available in Tk, but it's mix of power, flexibility and ease of use, along with its natural fit with today's layouts (that rely on alignment of widgets) make it the best choice for general use. There are other geometry managers: "pack" is also quite powerful, but harder to use and understand; "place" gives you complete control of positioning each element; we'll see even widgets like paned windows, notebooks, canvas and text can act as geometry managers.

Grid was first introduced to Tk in 1996, several years after Tk became popular, and took a while to catch on. Before that, developers had always used "pack" to do constraint-based geometry management. When grid came out, many developers kept using pack, and you'll still find it used in many Tk programs and documentation. While there's nothing technically wrong with it, the algorithm's behavior is often hard to understand. More importantly, because the order that widgets are packed is significant in determining layout, modifying existing layouts can be more difficult.

Grid has all the power of pack, generally produces nicer layouts (because it makes it easy to align widgets both horizontally and vertically), and is easier to learn and use. Because of that, we think grid is the right choice for most developers most of the time. Start your new programs using grid, and switch old ones to grid as you're making changes to an existing user interface.

The reference documentation for grid (http://www.tcl.tk/man/tcl8.6/TkCmd/grid.htm) provides an exhaustive description of grid, its behaviors and all options.

Columns and Rows

Using grid, widgets are assigned a "`column`" number and a "`row`" number, which indicates their relative position to each other. All widgets in the same column will, therefore, be above or below each other, while those in the same row will be to the left or right of each other.

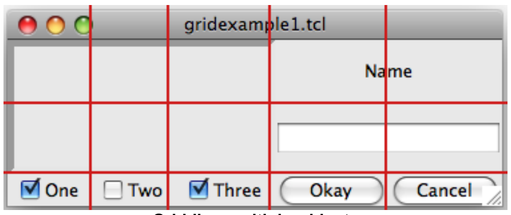
Column and row numbers must be integers, with the first column and row starting at 0. You can leave gaps in column and row numbers (e.g. column 0, 1, 2, 10, 11, 12, 20, 21), which is handy if you plan to add more widgets in the middle of the user interface at a later time.

The width of each column (or height of each row) depends on the width or height of the widgets contained within the column or row. This means when sketching out your user interface, and dividing it into rows and columns, you don't need to worry about each column or row being equal width.

Spanning Multiple Cells

Widgets can take up more than a single cell in the grid; to do this, you'll use the "`columnspan`" and "`rowspan`" options when gridding the widget. These are analogous to the "colspan" and "rowspan" attribute of HTML tables.

Here is an example of creating a user interface that has multiple widgets, some that take up more than a single cell.



Gridding multiple widgets

```
ttk::frame .c
ttk::frame .c.f -borderwidth 5 -relief sunken -width 200 -height 100
ttk::label .c.namelbl -text Name
ttk::entry .c.name
ttk::checkbutton .c.one -text One -variable one -onvalue 1; set one 1
ttk::checkbutton .c.two -text Two -variable two -onvalue 1; set two 0
ttk::checkbutton .c.three -text Three -variable three -onvalue 1; set three 1
ttk::button .c.ok -text Okay
ttk::button .c.cancel -text Cancel

grid .c -column 0 -row 0
grid .c.f -column 0 -row 0 -columnspan 3 -rowspan 2
grid .c.namelbl -column 3 -row 0 -columnspan 2
grid .c.name -column 3 -row 1 -columnspan 2
grid .c.one -column 0 -row 3
grid .c.two -column 1 -row 3
grid .c.three -column 2 -row 3
grid .c.ok -column 3 -row 3
grid .c.cancel -column 4 -row 3
```

```
require 'tk'
require 'tkextlib/tile'
root = TkRoot.new

content = Tk::Tile::Frame.new(root)
frame = Tk::Tile::Frame.new(content) {borderwidth 5; relief "sunken"; width 200; height 100}
namelbl = Tk::Tile::Label.new(content) {text "Name"}
name = Tk::Tile::Entry.new(content)
$option_one = TkVariable.new( 1 )
one = Tk::Tile::CheckButton.new(content) {text "One"; variable $option_one; onvalue 1}
$option_two = TkVariable.new( 0 )
two = Tk::Tile::CheckButton.new(content) {text "Two"; variable $option_two; onvalue 1}
$option_three = TkVariable.new( 1 )
three = Tk::Tile::CheckButton.new(content) {text "Three"; variable $option_three; onvalue 1}
ok = Tk::Tile::Button.new(content) {text "Okay"}
cancel = Tk::Tile::Button.new(content) {text "Cancel"}

content.grid :column => 0, :row => 0
frame.grid :column => 0, :row => 0, :columnspan => 3, :rowspan => 2
namelbl.grid :column => 3, :row => 0, :columnspan => 2
name.grid :column => 3, :row => 1, :columnspan => 2
one.grid :column => 0, :row => 3
two.grid :column => 1, :row => 3
three.grid :column => 2, :row => 3
ok.grid :column => 3, :row => 3
cancel.grid :column => 4, :row => 3

Tk.mainloop
```

```
use Tkx;

my $mw = Tkx::widget->new(".");
my $content = $mw->new_ttk__frame;
my $frame = $content->new_ttk__frame(-borderwidth => 5, -relief => "sunken", -width => 200, -height => 100);
my $namelbl = $content->new_ttk__label(-text => "Name");
my $name = $content->new_ttk__entry;
$option_one = 1; $option_two = 0; $option_three = 1;
my $one = $content->new_ttk__checkbutton(-text => "One", -variable => \$option_one, -onvalue => 1);
my $two = $content->new_ttk__checkbutton(-text => "Two", -variable => \$option_two, -onvalue => 1);
my $three = $content->new_ttk__checkbutton(-text => "Three", -variable => \$option_three, -onvalue => 1);
my $ok = $content->new_ttk__button(-text => "Okay");
my $cancel = $content->new_ttk__button(-text => "Cancel");

$content->g_grid(-column => 0, -row => 0);
$frame->g_grid(-column => 0, -row => 0, -columnspan => 3, -rowspan => 2);
$namelbl->g_grid(-column => 3, -row => 0, -columnspan => 2);
$name->g_grid(-column => 3, -row => 1, -columnspan => 2);
$one->g_grid(-column => 0, -row => 3);
$two->g_grid(-column => 1, -row => 3);
$three->g_grid(-column => 2, -row => 3);
$ok->g_grid(-column => 3, -row => 3);
$cancel->g_grid(-column => 4, -row => 3);

Tkx::MainLoop;
```

```
from tkinter import *
from tkinter import ttk

root = Tk()

content = ttk.Frame(root)
frame = ttk.Frame(content, borderwidth=5, relief="sunken", width=200, height=100)
namelbl = ttk.Label(content, text="Name")
name = ttk.Entry(content)

onevar = BooleanVar()
twovar = BooleanVar()
threevar = BooleanVar()
onevar.set(True)
twovar.set(False)
threevar.set(True)

one = ttk.Checkbutton(content, text="One", variable=onevar, onvalue=True)
two = ttk.Checkbutton(content, text="Two", variable=twovar, onvalue=True)
three = ttk.Checkbutton(content, text="Three", variable=threevar, onvalue=True)
ok = ttk.Button(content, text="Okay")
cancel = ttk.Button(content, text="Cancel")

content.grid(column=0, row=0)
frame.grid(column=0, row=0, columnspan=3, rowspan=2)
namelbl.grid(column=3, row=0, columnspan=2)
name.grid(column=3, row=1, columnspan=2)
one.grid(column=0, row=3)
two.grid(column=1, row=3)
three.grid(column=2, row=3)
ok.grid(column=3, row=3)
cancel.grid(column=4, row=3)

root.mainloop()
```

Layout within the Cell

Because the width of a column (and height of a row) depends on all the widgets that have been added to it, the odds are that at least some widgets will have a smaller width or height than has been allocated for the cell its been placed in. So the question becomes, where exactly should it be put within the cell?

By default, if a cell is larger than the widget contained in it, the widget will be centered within it, both horizontally and vertically, with the master's background showing in the empty space around it. The `"sticky"` option can be used to change this default behavior.

The value of the `"sticky"` option is a string of 0 or more of the compass directions "nsew", specifying which edges of the cell the widget should be "stuck" to. For example, a value of `"n"` (north) will jam the widget up against the top side, with any extra vertical space on the bottom; the widget will still be centered horizontally. A value of `"nw"` (north-west) means the widget will be stuck to the top left corner, with extra space on the bottom and right.

In Tkinter, you can also specify this as a list, containing any of N, S, E and W.

Specifying two opposite edges, such as `"we"` (west, east) means that the widget will be stretched, in this case, so it is stuck both to the left and right edge. So the widget will then be wider than its "ideal" size. Most widgets have options that can control how they are displayed if they are larger than needed. For example, a label widget has an `"anchor"` option which controls where the text of the label will be positioned.

If you want the widget to expand to fill up the entire cell, grid it with a sticky value of `"nsew"` (north, south, east, west) meaning it will stick to every side.

Handling Resize

If you've taken a peek below and added the extra "`sticky`" options to our example, when you try it out you'll notice things still don't look quite right (the entry is lower on the screen then we'd want), and things are even worse if you try to resize the window — nothing moves at all!

It looks like "sticky" may tell Tk *how* to react if the cell's row or column does resize, but doesn't actually say that the row or columns *should* resize if extra room becomes available. Let's fix that.

Every column and row has a "`weight`" grid option associated with it, which tells it how much it should grow if there is extra room in the master to fill. By default, the weight of each column or row is 0, meaning don't expand to fill space.

For the user interface to resize then, we'll need to give a positive weight to the columns we'd like to expand. This is done using the "`columnconfigure`" and "`rowconfigure`" methods of grid. If two columns have the same weight, they'll expand at the same rate; if one has a weight of 1, another of 3, the latter one will expand three pixels for every one pixel added to the first.

Both "`columnconfigure`" and "`rowconfigure`" also take a "`minsize`" grid option, which specifies a minimum size which you really don't want the column or row to shrink beyond.

Padding

Normally, each column or row will be directly adjacent to the next, so that widgets will be right next to each other. This is sometimes what you want (think of a listbox and its scrollbar), but often you want some space between widgets. In Tk, this is called padding, and there are several ways you can choose to add it.

We've already actually seen one way, and that is using a widget's own options to add the extra space around it. Not all widgets have this, but one that does is a frame; this is useful because frames are most often used as the master to grid other widgets. The frame's "`padding`" option lets you specify a bit of extra padding inside the frame, whether the same amount for each of the four sides, or even different for each.

A second way is using the "`padx`" and "`pady`" grid options when adding the widget. As you'd expect, "`padx`" puts a bit of extra space to the left and right of the widget, while "`pady`" adds extra space top and bottom. A single value for the option puts the same padding on both left and right (or top and bottom), while a two-value list lets you put different amounts on left and right (or top and bottom). Note that this extra padding is within the grid cell containing the widget.

If you want to add padding around an entire row or column, the "`columnconfigure`" and "`rowconfigure`" methods accept a "`pad`" option, which will do this for you.

Let's add the extra sticky, resizing, and padding behavior to our example (additions in bold).

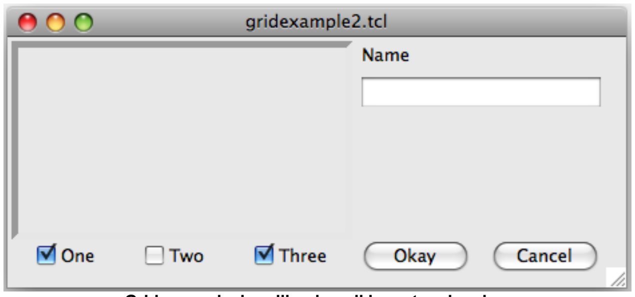
```
ttk::frame .c -padding "3 3 12 12"
ttk::frame .c.f -borderwidth 5 -relief sunken -width 200 -height 100
ttk::label .c.namelbl -text Name
ttk::entry .c.name
ttk::checkbutton .c.one -text One -variable one -onvalue 1; set one 1
ttk::checkbutton .c.two -text Two -variable two -onvalue 1; set two 0
ttk::checkbutton .c.three -text Three -variable three -onvalue 1; set three 1
ttk::button .c.ok -text Okay
ttk::button .c.cancel -text Cancel

grid .c -column 0 -row 0 -sticky nsew
grid .c.f -column 0 -row 0 -columnspan 3 -rowspan 2 -sticky nsew
grid .c.namelbl -column 3 -row 0 -columnspan 2 -sticky nw -padx 5
grid .c.name -column 3 -row 1 -columnspan 2 -sticky new -pady 5 -padx 5
grid .c.one -column 0 -row 3
grid .c.two -column 1 -row 3
grid .c.three -column 2 -row 3
grid .c.ok -column 3 -row 3
grid .c.cancel -column 4 -row 3

grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
grid columnconfigure .c 0 -weight 3
grid columnconfigure .c 1 -weight 3
grid columnconfigure .c 2 -weight 3
grid columnconfigure .c 3 -weight 1
grid columnconfigure .c 4 -weight 1
grid rowconfigure .c 1 -weight 1
```

```
require 'tk'
require 'tkextlib/tile'
root = TkRoot.new

content = Tk::Tile::Frame.new(root) {padding "3 3 12 12"}
frame = Tk::Tile::Frame.new(content) {borderwidth 5; relief "sunken"; width 200; height 100}
namelbl = Tk::Tile::Label.new(content) {text "Name"}
name = Tk::Tile::Entry.new(content)
$option_one = TkVariable.new( 1 )
one = Tk::Tile::CheckButton.new(content) {text "One"; variable $option_one; onvalue 1}
$option_two = TkVariable.new( 0 )
two = Tk::Tile::CheckButton.new(content) {text "Two"; variable $option_two; onvalue 1}
$option_three = TkVariable.new( 1 )
three = Tk::Tile::CheckButton.new(content) {text "Three"; variable $option_three; onvalue 1}
ok = Tk::Tile::Button.new(content) {text "Okay"}
cancel = Tk::Tile::Button.new(content) {text "Cancel"}

content.grid :column => 0, :row => 0, :sticky => 'nsew'
frame.grid :column => 0, :row => 0, :columnspan => 3, :rowspan => 2, :sticky => 'nsew'
namelbl.grid :column => 3, :row => 0, :columnspan => 2, :sticky => 'nw', :padx => 5
name.grid :column => 3, :row => 1, :columnspan => 2, :sticky => 'new', :pady => 5, :padx => 5
one.grid :column => 0, :row => 3
two.grid :column => 1, :row => 3
three.grid :column => 2, :row => 3
ok.grid :column => 3, :row => 3
cancel.grid :column => 4, :row => 3

TkGrid.columnconfigure( root, 0, :weight => 1 )
TkGrid.rowconfigure( root, 0, :weight => 1 )
TkGrid.columnconfigure( content, 0, :weight => 3 )
TkGrid.columnconfigure( content, 1, :weight => 3 )
TkGrid.columnconfigure( content, 2, :weight => 3 )
TkGrid.columnconfigure( content, 3, :weight => 1 )
TkGrid.columnconfigure( content, 4, :weight => 1 )
TkGrid.rowconfigure( content, 1, :weight => 1)

Tk.mainloop
```

```
use Tkx;

my $mw = Tkx::widget->new(".");
my $content = $mw->new_ttk__frame(-padding => "3 3 12 12");
my $frame = $content->new_ttk__frame(-borderwidth => 5, -relief => "sunken", -width => 200, -height => 100);
my $namelbl = $content->new_ttk__label(-text => "Name");
my $name = $content->new_ttk__entry;
$option_one = 1; $option_two = 0; $option_three = 1;
my $one = $content->new_ttk__checkbutton(-text => "One", -variable => \$option_one, -onvalue => 1);
my $two = $content->new_ttk__checkbutton(-text => "Two", -variable => \$option_two, -onvalue => 1);
my $three = $content->new_ttk__checkbutton(-text => "Three", -variable => \$option_three, -onvalue => 1);
my $ok = $content->new_ttk__button(-text => "Okay");
my $cancel = $content->new_ttk__button(-text => "Cancel");

$content->g_grid(-column => 0, -row => 0, -sticky => "nsew");
$frame->g_grid(-column => 0, -row => 0, -columnspan => 3, -rowspan => 2, -sticky => "nsew");
$namelbl->g_grid(-column => 3, -row => 0, -columnspan => 2, -sticky => "nw", -padx => 5);
$name->g_grid(-column => 3, -row => 1, -columnspan => 2, -sticky => "new", -pady => 5, -padx => 5);
$one->g_grid(-column => 0, -row => 3);
$two->g_grid(-column => 1, -row => 3);
$three->g_grid(-column => 2, -row => 3);
$ok->g_grid(-column => 3, -row => 3);
$cancel->g_grid(-column => 4, -row => 3);

$mw->g_grid_columnconfigure(0, -weight => 1);
$mw->g_grid_rowconfigure(0, -weight => 1);
$content->g_grid_columnconfigure(0, -weight => 3);
$content->g_grid_columnconfigure(1, -weight => 3);
$content->g_grid_columnconfigure(2, -weight => 3);
$content->g_grid_columnconfigure(3, -weight => 1);
$content->g_grid_columnconfigure(4, -weight => 1);
$content->g_grid_rowconfigure(1, -weight => 1);

Tkx::MainLoop;
```

```
from tkinter import *
from tkinter import ttk

root = Tk()

content = ttk.Frame(root, padding=(3,3,12,12))
frame = ttk.Frame(content, borderwidth=5, relief="sunken", width=200, height=100)
namelbl = ttk.Label(content, text="Name")
name = ttk.Entry(content)

onevar = BooleanVar()
twovar = BooleanVar()
threevar = BooleanVar()

onevar.set(True)
twovar.set(False)
threevar.set(True)

one = ttk.Checkbutton(content, text="One", variable=onevar, onvalue=True)
two = ttk.Checkbutton(content, text="Two", variable=twovar, onvalue=True)
three = ttk.Checkbutton(content, text="Three", variable=threevar, onvalue=True)
ok = ttk.Button(content, text="Okay")
cancel = ttk.Button(content, text="Cancel")

content.grid(column=0, row=0, sticky=(N, S, E, W))
frame.grid(column=0, row=0, columnspan=3, rowspan=2, sticky=(N, S, E, W))
namelbl.grid(column=3, row=0, columnspan=2, sticky=(N, W), padx=5)
name.grid(column=3, row=1, columnspan=2, sticky=(N, E, W), pady=5, padx=5)
one.grid(column=0, row=3)
two.grid(column=1, row=3)
three.grid(column=2, row=3)
ok.grid(column=3, row=3)
cancel.grid(column=4, row=3)

root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)
content.columnconfigure(0, weight=3)
content.columnconfigure(1, weight=3)
content.columnconfigure(2, weight=3)
content.columnconfigure(3, weight=1)
content.columnconfigure(4, weight=1)
content.rowconfigure(1, weight=1)

root.mainloop()
```

This looks more promising. Play around with the example to get a feel for the resize behavior.



Grid example, handling in-cell layout and resize.

You'll notice the little resize gadget at the very bottom right of the window; while we're just taking the easy route and avoiding it with the extra padding, we'll see later how to better take it into account using a "sizegrip" widget.

Additional Grid Features

As you could see from the grid reference (http://www.tcl.tk/man/tcl8.6/TkCmd/grid.htm), there are lots of other things you can do with grid. Here are a few of the more useful ones.

Querying and Changing Grid Options

Like widgets themselves, it's easy to introspect the various grid options, as well as change them; setting options when you first grid the widget is just a convenience, and you can certainly change them anytime you'd like.

The `"slaves"` method will tell you all the widgets that have been gridded inside a master, or optionally those within just a certain column or row. The `"info"` method will give you a list of all the grid options for a widget and their values. Finally, the `"configure"` method lets you change one or more grid options on a widget.

These are illustrated in this interactive session:

```
% grid slaves .c
.c.cancel .c.ok .c.three .c.two .c.one .c.name .c.namelbl .c.f
% grid slaves .c -row 3
.c.cancel .c.ok .c.three .c.two .c.one
% grid slaves .c -column 0
.c.one .c.f
% grid info .c.namelbl
-in .c -column 3 -row 0 -columnspan 2 -rowspan 1 -ipadx 0 -ipady 0 -padx 5 -pady 0 -sticky nw
% grid configure .c.namelbl -sticky ew
% grid info .c.namelbl
-in .c -column 3 -row 0 -columnspan 2 -rowspan 1 -ipadx 0 -ipady 0 -padx 5 -pady 0 -sticky ew
```

```
>> TkGrid.slaves(content, nil)
=> [#<Tk::Tile::TButton:0x380fa4 @path=".w00000.w00008">,
#<Tk::Tile::TButton:0x500d20 @path=".w00000.w00007">,
#<Tk::Tile::TCheckButton:0x50aac8 @path=".w00000.w00006">,
#<Tk::Tile::TCheckButton:0x524158 @path=".w00000.w00005">,
#<Tk::Tile::TCheckButton:0x54b4b0 @path=".w00000.w00004">,
#<Tk::Tile::TEntry:0x5d07a0 @path=".w00000.w00003">,
#<Tk::Tile::TLabel:0x5fa9b0 @path=".w00000.w00002">,
#<Tk::Tile::TFrame:0x5ff280 @visual=nil, @container=nil, @colormap=nil,
@path=".w00000.w00001", @db_class=Tk::Tile::TFrame, @classname="TFrame">]
>> TkGrid.slaves(content, :row => 3)
=> [#<Tk::Tile::TButton:0x380fa4 @path=".w00000.w00008">,
#<Tk::Tile::TButton:0x500d20 @path=".w00000.w00007">,
#<Tk::Tile::TCheckButton:0x50aac8 @path=".w00000.w00006">,
#<Tk::Tile::TCheckButton:0x524158 @path=".w00000.w00005">,
#<Tk::Tile::TCheckButton:0x54b4b0 @path=".w00000.w00004">]
>> TkGrid.slaves(content, :column => 0)
=> [#<Tk::Tile::TCheckButton:0x54b4b0 @path=".w00000.w00004">,
#<Tk::Tile::TFrame:0x5ff280 @visual=nil, @container=nil, @colormap=nil,
@path=".w00000.w00001", @db_class=Tk::Tile::TFrame, @classname="TFrame">]
>> TkGrid.info(namelbl)
=> {"ipadx"=>0, "ipady"=>0, "columnspan"=>2, "row"=>0, "column"=>3,
"in"=>#<Tk::Tile::TFrame:0x60ef78 @visual=nil, @container=nil, @colormap=nil,
 @path=".w00000", @db_class=Tk::Tile::TFrame, @classname="TFrame">, "rowspan"=>1,
"sticky"=>"nw", "padx"=>5, "pady"=>0}
>> TkGrid.configure(namelbl, :sticky => 'ew')
=> ""
>> TkGrid.info(namelbl)
=> {"ipadx"=>0, "ipady"=>0, "columnspan"=>2, "row"=>0, "column"=>3,
"in"=>#<Tk::Tile::TFrame:0x60ef78 @visual=nil, @container=nil, @colormap=nil,
@path=".w00000", @db_class=Tk::Tile::TFrame, @classname="TFrame">, "rowspan"=>1,
"sticky"=>"ew", "padx"=>5, "pady"=>0}
```

```
Perl> $content->g_grid_slaves
.f.b2 .f.b .f.c3 .f.c2 .f.c .f.e .f.l .f.f
Perl> $content->g_grid_slaves(-row => 3)
.f.b2 .f.b .f.c3 .f.c2 .f.c
Perl> $content->g_grid_slaves(-column => 0)
.f.c .f.f
Perl> $namelbl->g_grid_info
-in .f -column 3 -row 0 -columnspan 2 -rowspan 1 -ipadx 0 -ipady 0 -padx 5 -pady 0 -sticky nw
Perl> $namelbl->g_grid_configure(-sticky => "ew")

Perl> $namelbl->g_grid_info
-in .f -column 3 -row 0 -columnspan 2 -rowspan 1 -ipadx 0 -ipady 0 -padx 5 -pady 0 -sticky ew
```

```
>>> content.grid_slaves()
<map object at 0x00C3F470>
>>> for w in content.grid_slaves(): print(w)
...
.14597008.14622128
.14597008.14622096
.14597008.14622064
.14597008.14622032
.14597008.14622000
.14597008.14621872
.14597008.14621840
.14597008.14621808
>>> for w in content.grid_slaves(row=3): print(w)
...
.14597008.14622128
.14597008.14622096
.14597008.14622064
.14597008.14622032
.14597008.14622000
>>> for w in content.grid_slaves(column=0): print(w)
...
.14597008.14622000
.14597008.14621808
>>> namelbl.grid_info()
{'rowspan': '1', 'column': '3', 'sticky': 'nw', 'ipady': '0', 'ipadx': '0', 'columnspan': '2',
'in': <tkinter.ttk.Frame object at 0x00DEBB90>, 'pady': '0', 'padx': '5', 'row': '0'}
>>> namelbl.grid_configure(sticky=(E,W))
>>> namelbl.grid_info()
{'rowspan': '1', 'column': '3', 'sticky': 'ew', 'ipady': '0', 'ipadx': '0', 'columnspan': '2',
'in': <tkinter.ttk.Frame object at 0x00DEBB90>, 'pady': '0', 'padx': '5', 'row': '0'}
```

Internal Padding

You saw how the "`padx`" and "`pady`" grid options added extra space around the outside of a widget. There's also a less used type of padding called "internal padding", which is controlled by the grid options "`ipadx`" and "`ipady`".

The difference can be subtle. Let's say you have a frame that's 20x20, and specify normal (external) padding of 5 pixels on each side. The frame will request a 20x20 rectangle (its natural size) from the geometry manager. Normally, that's what it will be granted, so it'll get a 20x20 rectangle for the frame, surrounded by a 5-pixel border.

With internal padding, the geometry manager will effectively add the extra padding to the widget when figuring out its natural size, as if the widget has requested a 30x30 rectangle. If the frame is centered, or attached to a single side or corner (using "`sticky`"), you'll end up with a 20x20 frame with extra space around it. If however the frame is set to stretch (i.e. a "`sticky`" value of "`we`", "`ns`", or "`nwes`") it will fill the extra space, resulting in a 30x30 frame, with no border.

Forget and Remove

The "`forget`" method of grid, taking as arguments a list of one or more slave widgets, can be used to remove slaves from the grid they're currently part of. This does not destroy the widget altogether, but takes it off the screen, as if it had not been gridded in the first place. You can grid it again later, though any grid options you'd originally assigned will have been lost.

The "`remove`" method of grid works the same, except that the grid options will be remembered.

Nested Layouts

As your user interface gets more complicated, the grid that you're using to organize all your widgets can get more and more complicated, and more fine-grained. This can make changing and maintaining your program very difficult.

Luckily, you don't have to manage your entire user interface with a single grid. If you have one area of your user interface that is fairly independent of others, create a new frame to hold that area, and grid the widgets that are part of that area within that frame. If you had a graphics program of some kind with multiple palettes, toolbars, and so on, each one of those areas might be a candidate for putting in its own frame.

In theory, these frames, each with its own grid, can be nested arbitrarily deep, though in practice this usually doesn't go beyond a few levels. This can be a big help in modularizing your program. If, for example, you have a palette of drawing tools, you can create the whole thing in a separate procedure, including creating all the component widgets, gridding them together, setting up event bindings, and so on. From the point of view of your main program, all it needs to see is the single frame widget containing it all.

Our examples have shown just a hint of this, where a content frame was gridded into the main window, and then all the other widgets gridded into the content frame.

Previous: Basic Widgets (widgets.html) | Contents (index.html) | Single Page (onepage.html) | Next: More Widgets (morewidgets.html)