(U) The `timedelta` type is a measure of duration between two time events. So, if we subtract two datetime objects (or `date` or `time`) as we did above, we get a `timedelta` object. The properties of a `timedelta` object are `(days, seconds, microseconds, milliseconds, minutes, hours, weeks)`. They are all optional and set to 0 by default. A `timedelta` object can take these values as arguments but converts and normalizes the data into days, seconds, and microseconds.

```
from datetime import timedelta
```

```
day = timedelta(days=1)
```

```
day
```

```
now = datetime.datetime.now()
```

```
now + day
```

```
now - day
```

```
now + 300*day
```

```
now - 175*day
```

```
year = timedelta(days=365)
```

```
another_year = timedelta(weeks=40, days=84, hours=23, minutes=50, seconds=600)
```

```
year == another_year
```

```
year.total_seconds()
```

```
ten_years = 10*year
ten_years
```

(U) Conversions

(U) It's easy to get confused when attempting to convert back and forth between strings, numbers, `time` s, and `datetime` s. When you need to do it, the best course of action is probably to open up an interactive session, fiddle around until you have what you need, then capture that in a well-named function. Still, some pointers may be helpful.

(U) Objects of types **time** and **datetime** provide `strptime` and `strftime` methods for converting times and dates from strings (a.k.a. ***p***arsing) and converting to strings (a.k.a. ***f***ormatting), respectively. These methods employ a custom syntax that is shared across many programming languages.

```
print(ga_dt)
```

(U) Localize the time using the Georgia timezone, then convert to Kabul timezone

```
kabul_tz = pytz.timezone('Asia/Kabul')
kabul_dt = ga_tz.localize(ga_dt).astimezone(kabul_tz)
print(kabul_tz.normalize(kabul_dt))
```

(U) To get a list of all timezones:

```
pytz.all_timezones
```

(U) The arrow package

(U) The arrow package is a third party package useful for manipulating dates and times in a non-naive way (in this case, meaning that it deals with multiple timezones very seamlessly). Examples below are based on examples from "20 Python Libraries You Aren't Using (But Should)" on Safari.

```
import ipydeps
ipydeps.pip('arrow')
import arrow
```

```
t0 = arrow.now()
print(t0)

t1 = arrow.utcnow()
print(t1)

difference = (t0 - t1).total_seconds()

print('Total difference: %.2f seconds' % difference)
```

```
t0 = arrow.now()
t0
```

```
t0.date()
```

```
t0.time()
```

```
t0.timestamp
```

```
t0.year
```

```
t0.month
```

```
t0.day
```

```
t0.datetime
```

```
t1.datetime
```

```
t0 = arrow.now()
t0.humanize()
```

```
t0.humanize()
```

```
t0 = t0.replace(hours=-3,minutes=10)
t0.humanize()
```

(U) The parsedate module

(U) The parsedate package is a third party package that does a very good job of parsing dates and times from "messy" input formats. It also does a good job of calculating relative times from human-sytle inpu. Examples below are from "20 Python Libraries You Aren't Using (But Should)" on Safari.

```
ipydeps.pip('parsedatetime')
import parsedatetime as pdt
```

Doc ID: 6689695

```
cal = pdt.Calendar()

examples = [
    "2016-07-16",
    "2016/07/16",
    "2016-7-16",
    "2016/7/16",
    "07-16-2016",
    "7-16-2016",
    "7-16-16",
    "7/16/16",
]

#print the header
print('{:30s}{:>30s}'.format('Input', 'Result'))
print('=' * 60)

#loop through the examples list and show that parseDT successfully parses out the date/time based on the messy values
for e in examples:
    dt, result = cal.parseDT(e)
    print('{:<30s}{:>30}'.format('"' + e + '"', dt.ctime()))
```

```
examples = [
    "19 November 1975",
    "19 November 75",
    "19 Nov 75",
    "tomorrow",
    "yesterday",
    "10 minutes from now",
    "the first of January, 2001",
    "3 days ago",
    "in four days' time",
    "two weeks from now",
    "three months ago",
    "2 weeks and 3 days in the future",
]

#print the time right now for reference
print('Now: {}'.format(datetime.datetime.now().ctime()), end='\n\n')

#print the header
print('{:40s}{:>30s}'.format('Input', 'Result'))
print('=' * 70)

#loop through the examples list to show how parseDT can successfully determine the date/time based on both messy input
# and messy relative time offset inputs
for e in examples:
    dt, result = cal.parseDT(e)
    print('{:<40s}{:>30}'.format('"' + e + '"', dt.ctime()))
```

COMP3321 Datetime Exercises

0

(b)(3)-P.L. 86-36

Created almost 3 years ago by [redacted] in COMP 3321

3 143 23

python exercises

(U) Datetime Exercises for COMP3321

Recommendations

(U) Datetime Exercise

(U) How long before Christmas?

(U) How many seconds since you were born?

(U) What is the average number of days between Easter and Christmas for the years 2000 - 2999?

(U) What day of the week does Christmas fall on this year?

(U) You get a intercepted email with a POSIX timestamp of 1435074325. The email is from the leader of a Zendian extremist group and says that there will be an attack on the Zendian capitol in 14 hours. In Zendian local time, when will the attack occur? (Assume Zendia is in the same time zone as Kabul)

Module: Interactive User Input with ipywidgets

0

(b)(3)-P.L. 86-36

Updated almost 2 years ago by [redacted] in COMP 3321

3 3 393 200

fcx93 ipywidgets python tutorial input

(U) Covers the ipywidgets library for getting interactive user input in Jupyter

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U) ipywidgets

(U) ipywidgets is used for making interactive widgets inside your jupyter notebook

(U) The most basic way to get user input is to use the python built in `input` function. For more complicated types of interaction, you can use ipywidgets

```
#input example (not using ipywidgets)
a=input("Give me your input: ")
print("your input was: "+a)
```

```
import ipywidgets
from ipywidgets import *
```

`interact` is the easiest way to get started with ipywidgets by creating a user interface and automatically calling the specified function

```
def f(x):
    return x*2

interact(f,x=10)
```

```
def g(check,y):
    print("{} {}".format(check,y))
interact(g,check=True,y="Hi there!")
```

But, if you need more flexibility, you can start from scratch by picking a widget and then calling the functionality you want. Hint: you get more widget choices this way.

```
IntSlider()
```

```
w=IntSlider()
```

```
w
```

You can explicitly display using IPython's display module. Note what happens when you display the same widget more than once!

```
from IPython.display import display
display(w)
```

```
w.value
```

Now we have a value from our slider we can use in code. But what other attributes or "keys" does our slider widget have?

```
w.max
```

```
new_w=IntSlider(max=200)
display(new_w)
```

You can also close your widget

```
w.close()
new_w.close()
```

Here are all the available widgets:

```
Widget.widget_types
```

Categories of widgets

Numeric: IntSlider, FloatSlider, IntRangeSlider, FloatRangeSlider, IntProgress, FloatProgress, BoundedIntText, BoundedFloatText, IntText, FloatText

Boolean: ToggleButton, Checkbox, Valid

Selection: Dropdown, RadioButtons, Select, ToggleButtons, SelectMultiple

String Widgets: Text, Textarea

Other common: Button, ColorPicker, HTML, Image

```
Dropdown(options=["1","2","3","cat"])
```

```
bt = Button(description="Click me!")
display(bt)
```

Buttons don't do much on their own, so we have to use some event handling. We can define a function with the desired behavior and call it with the buttons `on_click` method.

```
def clicker(b):
    print("Hellow World!!!!")

bt.on_click(clicker)
```

```
def f(change):
    print(change['new'])

w = IntSlider()
display(w)
w.observe(f,names='value')
```

Wrapping Multiple Widgets in Boxes

When working with multiple input widgets, it's often nice to wrap it all in a nice little box. `ipywidgets` provides a few options for this--we'll cover `HBox` (horizontal box) and `VBox` (vertical box).

HBox

This will display the widgets horizontally

```
fruit_list = Dropdown(
  options = ['apple', 'cherry', 'orange', 'plum', 'pear']
)
fruit_label = HTML(
  value = 'Select a fruit from the list:  '
)
fruit_box = HBox(children=[fruit_label, fruit_list])
fruit_box
```

VBox

This will display the widgets (or boxes) vertically

```
num_label = HTML(
  value = 'Choose the number of fruits:  '
)

num_options = IntSlider(
  min=1,
  max=20
)

num_box = HBox(children=(num_label, num_options))

type_label = HTML(
  value = 'Select the type of fruit:  '
)

type_options = RadioButtons(
  options=('Under-ripe', 'Ripe', 'Rotten')
)

type_box = HBox(children=(type_label, type_options))


fruit_vbox = VBox(children=(fruit_box, num_box, type_box))
fruit_vbox
```

Specify Layout of the Widgets/Boxes

```
form_item_layout = Layout(
  display='flex',
  flex_flow='row',
  justify_content='space-between',
  width='70%',
  align_items='initial',
)

veggie_label = HTML(
  value = 'Select a vegetable from the list:  ',
  layout=Layout(width='20%', height='65px')
)

veggie_options = Dropdown(
  options=['corn', 'lettuce', 'tomato', 'potato', 'spinach'],
  layout=Layout(width='30%', height='65px')
)

veggie_box = HBox(children=(veggie_label, veggie_options),
                  layout=Layout(width='100%', border='solid 1px',
                                height='100px'))
veggie_box
```

Retrieving Values from a Box

```
box_values = {}
# the elements in a box can be accessed using the children attribute
for index, box in enumerate(fruit_vbox.children):
  for child in box.children:
    if type(child) != ipywidgets.widgets.widget_string.HTML:
      if index == 0:
        print("The selected fruit is: ", child.value)
        box_values['fruit'] = child.value
      elif index == 1:
        print("The select number of fruits is: ", str(child.value))
        box_values['count'] = child.value
      elif index == 2:
        print("The selected type of fruit is: ", str(child.value))
        box_values['type'] = child.value
box_values
```

Module: GUI Basics with Tkinter

(b)(3)-P.L. 86-36

0

Updated almost 2 years ago by [redacted] in COMP 3321

3 5 592 225

python fcs6 comp3321

(U) Module: GUI Basics with Tkinter

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U) Tkinter

(U) Tkinter comes as part of Python, so is readily available for use--just import it. **Note**: While Tkinter is almost always available, Python can be installed without it, e.g. if Tcl/Tk is not available when Python is compiled. Tk is a widget library that was originally designed for the Tcl scripting language, but now has been ported to Perl, Ruby, Python, C++ and more.

(U) **NOTE**: In Python 2 it must be used as `Tkinter` with a capital `T`.

(U) Setup

P.L. 86-36

(U//~~FOUO~~) This lesson cannot currently be run from Jupyter on LABBENCH due to displayback limitations. It should work using Anaconda's Jupyter locally [redacted] The examples can also be copied to files and run as scripts from MACHINESHOP if your display is properly configured.

(U//~~FOUO~~) On MACHINESHOP

(U//~~FOUO~~) To display Python Tk objects back from MACHINESHOP, you will need to run the following on your MASH instance. [redacted]

[redacted] **NOTE**: Not all steps may be necessary; verification needed.

```
yum -y groupinstall desktop
yum -y install tigervnc-server
yum -y install xrdp
/sbin/service xrdp start
chkconfig xrdp on
/sbin/service iptables stop
```

(U) What's a GUI (Graphical User Interface)?

(U) We all use them, some of us love them and hate them. Do we consider it 2- or 3-dimensional (2.5-dimensional)? Let's look at some very basic examples.

(U) Example 1

```
import tkinter as tk
root = tk.Tk()
root.mainloop()
```

We just created our first gui! But it doesn't do a whole lot yet. That is because we only created a blank/empty window that is waiting for our creation. `tk.Tk()` is the top level window that we will create for every gui that we make.

Parts of a gui:
Choose widgets ==> Arrange in window ==> Add functionality

(U) Example 2

A first look at widgets!

```
#Basic gui with a label and a button

import tkinter as tk
root = tk.Tk()

label = tk.Label(root, text="I am a label widget") #Create label
button = tk.Button(root, text="I am a button")     #create button

label.pack()                                       #Add label to gui
button.pack()                                      #Add button to gui
root.mainloop()
```

(U) Widget Types

Type	Description
`Button`	Users click on buttons to trigger some action. Button clicks can be translated into actions taken by your program. `Button` s usually display text but can show graphics.
`Canvas`	A surface on which you can draw graphs and/or plots and also use as the basis of your own widgets.
`CheckButton`	A special type of `Button` that has two states; clicking changes the state of the button from one to the other.
`Entry`	Used to enter single lines of text and all kinds of input.
Frame	A container for other widgets. One can set the border and background color and place other widgets in it.
`Label`	Used to display pieces of text or images, usually ones that won't change during the execution of the application.

Type	Description
`Listbox`	Used to display a set of choices. The user can select a single item or multiple items from the list. The `Listbox` can be also rendered as a set of radio buttons or checkboxes.
`Message`	Similar to `Text` but can automatically wrap text to a particular width and height.
`Menu`	Used to put a menu in your window if you need it. It corresponds to the menu bar at the top but can also be used as a pop-up.
`Menubutton`	Adds choices to your `Menu` s.
`Radiobutton`	Represents one of a set of mutually exclusive choices. Selecting one `Radiobutton` from a set deselects any others.
`Scale`	Lets the user set numeric values by dragging a slider.
`Scrollbar`	Implements scrolling on a larger widget such as a `Canvas` , `Listbox` , or `Text` .
`Text`	A multi-line formatted text widget that allows the textual content to be "rich." It may also contain embedded images and `Frame` s.
Toplevel	A special kind of `Frame` that interacts directly with the window manager. `Toplevel` s will usually have a title bar and features to interact with the window manager. The windows you see on your screen are mostly `Toplevel` windows, and your application can create additional `Toplevel` windows if it is set to do that.

Other widgets: `OptionMenu` , `LabelFrame` , `PanedWindow` , `Bitmap Class` , `Spinbox` , `Image Class` .

(U) Example 3

Let's look at some other widget examples.
Also notice, that widgets have their own special "widget variables." Instead of using builtin python types, tk widgets use their own objects for storing this internal information. The tk widget variables are: `StringVar` , `IntVar` , `DoubleVar` , and `BooleanVar`

Doc ID: 6689695

```
import tkinter as tk

root = tk.Tk()

tk.Label(root, text="Enter your Password:").pack()
tk.Button(root, text="Search").pack()

v = tk.IntVar()
tk.Checkbutton(root, text="Remember Me", variable=v).pack()
tk.Entry(root, width=30)

v2 = tk.IntVar()
tk.Radiobutton(root, text="Male", variable=v2, value=1).pack()
tk.Radiobutton(root, text="Female", variable=v2, value=2).pack()

var = tk.IntVar()
tk.OptionMenu(root, var, "Select Country", "USA", "UK", "India", "Others").pack()
tk.Scrollbar(root, orient='vertical').pack()

root.mainloop()
```

(U) Three ways to configure a widget

1. (U) Setting the values during initialization. (The way we have been doing it so far).
2. (U) Using keys to set the values.
3. (U) Using the widget's `configure` method.

(U) Widget Attributes

(U) There are tons of options that can be set, but here are a few of the important ones.

Attribute	Description
`background / bg`	The color of the body of the widget (e.g. `'red'`, `'blue'`, `'green'`, `'black'`).
`foreground / fg`	The color used for text.
`padx, pady`	The amount of padding to put around the widget horizontally and vertically. Without these the widget will be just big enough for its content.
`borderwidth`	Creates a visible border around a widget.
`height, width`	Specifies the height and width of the widget.
`disableforeground`	When a widget is disabled, this is the color of its text (usually gray).
`State`	Default is `'normal'` but also can use `'disabled'` or `'active'`

(U) Example 4

Let's try redoing Example 2 using the key/value method to make our label and the `.configure()` method to make our button.

```
#Basic gui with a label and a button

import tkinter as tk
root = tk.Tk()

label = tk.Label(root)
label["text"]="I am a label widget"        #using keys
button = tk.Button(root)
button.configure(text="I am a button")     #using configure

label.pack()                                    #Add label to gui
button.pack()                                   #Add button to gui
root.mainloop()
```

(U) Geometry Managers

(U) Now that we know how to create widgets, we're on to step two: arranging them in our window!

(U) There are mainly two types of geometry managers:

- `Pack`
- `Grid`

(U) There is also a third--Place--but maybe that's not for today.

(U) Pack

(U) Quick, easy, effective. If things get complicated, use `Grid` instead.

Attribute	Description
`fill`	Can be `X`, `Y`, or `Both`. `X` does the horizontal, `Y` does the vertical.
`expand`	`False` means the widget is never resized, `True` means the widget is resized when the container is resized.
`side`	Which side the widget will be packed against (`TOP`, `BOTTOM`, `RIGHT` or `LEFT`).

(U) Example 5

The pack geometry manager arranges widgets relative to window/frame you are putting them in. For example, if you select `side=LEFT` it will pack you widget against the left side of the widget.

```
#Example using pack geomtry manager

from tkinter import *
root = Tk()

parent = Frame(root)
# placing widgets top-down
Button(parent, text='ALL IS WELL').pack(fill=X)
Button(parent, text='BACK TO BASICS').pack(fill=X)
Button(parent, text='CATCH ME IF U CAN').pack(fill=X)
# placing widgets side by side
Button(parent, text='LEFT').pack(side=LEFT)
Button(parent, text='CENTER').pack(side=LEFT)
Button(parent, text='RIGHT').pack(side=LEFT)
parent.pack()
root.mainloop()
```

(U) Example 6

Generally, the pack geometry manager is best for simple gui's, but one way to make more complicated gui's using `pack` is to group widgets together in a `Frame` and then add the `Frame` to your window.

```
#Example using pack geometry manager

from tkinter import *
root = Tk()

frame = Frame(root)      #Add frame for grouping widgits

# demo of side and fill options
Label(frame, text="Pack Demo of side and fill").pack()
Button(frame, text="A").pack(side=LEFT, fill=Y)
Button(frame, text="B").pack(side=TOP, fill=X)
Button(frame, text="C").pack(side=RIGHT, fill=NONE)
Button(frame, text="D").pack(side=TOP, fill=BOTH)
frame.pack()
# note the top frame does not expand nor does it fill in

# X or Y directions
# demo of expand options - best understood by expanding the root widget and seeing the effect on all the three buttons below.
Label (root, text="Pack Demo of expand").pack()
Button(root, text="I do not expand").pack()
Button(root, text="I do not fill x but I do not expand").pack(expand = 1)
Button(root, text="I fill x and expand").pack(fill=X, expand=1)
root.mainloop()
```

(U) Grid

(U) Use it when things get complicated, but it can be complicated in itself!

Attribute	Description
`row`	The row in which the widget should appear.
`column`	The column in which the widget should appear.
`sticky`	Can be `N`, `S`, `E`, or `W`. One needs to actually see this work, but it's important if you want the widget to resize with everything else.
`rowspan, columnspan`	Widgets can start in one widget and occupy more than one row or column.

(U) Rowconfigure and columnconfigure

(U) Most layout definitions start with these.

Option	Description
`minsize`	Defines the row's or column's minimum size.
`pad`	Sets the size of the row or column by adding the specified amount of padding to the height of the row or the width of the column.
`weight`	Determines how additional space is distrbuted between the rows and columns as the frame expands. The higher the weight, the more of the additional space is taken up. A row weight of 2 will expand twice as fast as that of 1.

(U) Example 7

The `grid` geometry manager starts with row zero and column zero up in the top left hand corner of your window. When you add a widget using `grid`, the default is `row=0` and `column=0` so you don't have to explicitly state it, although it is good practice.

```
#Basic example using grid geometry manager

from tkinter import *
root = Tk()
Label(root, text="Username").grid(row=0, sticky=W)
Label(root, text="Password").grid(row=1, sticky=W)
Entry(root).grid(row=0, column=1, sticky=E)
Entry(root).grid(row=1, column=1, sticky=E)
Button(root, text="Login").grid(row=2, column=1, sticky=E)
root.mainloop()
```

(U) Example 8

You could create this example using `pack` ... But it you would probably need a lot of frames. Using `grid` for something like this is much easier!

```
#More advanced example using geometry manager

from tkinter import *
parent = Tk()
parent.title('Find & Replace')  #Title of window

#First label and text entry widgits
Label(parent, text="Find:").grid(row=0, column=0, sticky='e')
Entry(parent, width=60).grid(row=0, column=1, padx=2, pady=2, sticky='we', columnspan=9)

#second label and text entry widgits
Label(parent, text="Replace:").grid(row=1, column=0, sticky='e')
Entry(parent).grid(row=1, column=1, padx=2, pady=2, sticky='we', columnspan=9)

#buttons
Button(parent, text="Find").grid(
row=0, column=10, sticky='e' + 'w', padx=2, pady=2)
Button(parent, text="Find All").grid(
  row=1, column=10, sticky='e' + 'w', padx=2)
Button(parent, text="Replace").grid(row=2, column=10, sticky='e' + 'w', padx=2)
Button(parent, text="Replace All").grid(
  row=3, column=10, sticky='e' + 'w', padx=2)

#Checkboxes
Checkbutton(parent, text='Match whole word only').grid(
  row=2, column=1, columnspan=4, sticky='w')
Checkbutton(parent, text='Match Case').grid(
  row=3, column=1, columnspan=4, sticky='w')
Checkbutton(parent, text='Wrap around').grid(
  row=4, column=1, columnspan=4, sticky='w')

#Label and radio buttons
Label(parent, text="Direction:").grid(row=2, column=6, sticky='w')
Radiobutton(parent, text='Up', value=1).grid(
  row=3, column=6, columnspan=6, sticky='w')
Radiobutton(parent, text='Down', value=2).grid(
  row=3, column=7, columnspan=2, sticky='e')

#run gui
parent.mainloop()
```

(U) Object Oriented GUI's

Up until now, all our examples show how to use tkinter with out creating classes, but in reality, GUI programs usually get very large very quickly. To keep things organized, it's best to encapsulate your GUI in a class and group your widget creation inside of class functions.

(U) Example 9

Wrapping a GUI inside a class. Note that in real life, you probably won't be running a GUI from inside Jupyter. This example shows how to check if you are in `main`, which you will need if you are running from the command line.

```
from tkinter import *


class Application(Frame):
    '''
    class Application :: Basic Tkinter example
    '''

def create_widgets(self):
        #Create Widget
        self.hi_there = Button(self)
        self.hi_there['text'] = 'hello'
        self.hi_there['fg'] = 'blue'

        #Arrange widget
        self.hi_there.pack({'side':'left'})


        #Create Widget
        self.QUIT = Button(self)
        self.QUIT['text'] = 'Quit'
        self.QUIT['fg'] = 'red'

        #Arrange widget
        self.QUIT.pack({'side':'left'})


    def __init__(self, master = None):
        '''
        Constructor
        '''
        Frame.__init__(self,master)
        self.pack()
        self.create_widgets()


def main():
    root = Tk()
    app = Application(master=root)
    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Callbacks and Eventing

(U)Most widgets have a `command` attribute to associate a callback function for when the widget is clicked.

When you need your gui to respond to something other than a mouse click or a specific kind of mouse click, you can bind you widget to an event.

(U) Example 9.1

Add a simple callback function to example 9 using.

What does the `print` command do when you click the "hello" button?

What happens when you click the "Quit" button?

Hint: you really are "quitting" your program. It just doesn't destroy (that is, close) your window!

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    class Application :: Basic Tkinter example
    '''

    #Callback function
    def say_hello(self):
        print('Hello There')

    def create_widgets(self):
        #Create Widget
        self.hi_there = Button(self)
        self.hi_there['text'] = 'hello'
        self.hi_there['fg'] = 'blue'

        #Arrange widget
        self.hi_there.pack({'side':'left'})
        #call back functionality
        self.hi_there['command'] = self.say_hello


        #Create Widget
        self.QUIT = Button(self)
        self.QUIT['text'] = 'Quit'
        self.QUIT['fg'] = 'red'

        #Arrange widget
        self.QUIT.pack({'side':'left'})
        #call back functionality
        self.QUIT['command'] = self.quit


    def __init__(self, master = None):
        '''
        Constructor
        '''
        Frame.__init__(self,master)
        self.pack()
        self.create_widgets()


def main():
    root = Tk() #create window
    app = Application(master=root)
    app.mainloop()
```

```
if __name__ == '__main__':
    main()
```

(U) Example 9.2

(U) Using the key/value method made it easy to check we were following the pattern:

1. Choose widget
2. Add to window/arrange
3. Add functionality

But it is shorter to write the code using the initialize method. Even if your code isn't written in the order of these steps, this is still the order you want to think about them.

```
from tkinter import *


class Application(Frame):
    '''
    class Application :: Basic Tkinter example
    '''
    def say_hello(self):
        print('Hello There')

    def create_widgets(self):
        self.hi_there = Button(self, text='hello',fg='blue',command=self.say_hello)
        self.hi_there.pack(side='left')

        self.QUIT = Button(self,text='quit',fg='red',command=self.quit)
        self.QUIT.pack(side='left')

    def __init__(self, master = None):
        '''
        Constructor
        '''
        Frame.__init__(self,master)
        self.pack()
        self.create_widgets()


def main():
    root = Tk()
    app = Application(master=root)
    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Example 10

(U) Now that we have covered the three basics of GUI's: selecting widgets, arranging them in our window, and adding functionality, let's try a more complicated example.

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #---------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()


    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #------------------------------------------------------------

    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
```

```
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


def main():
    root=Tk()
    #set the size of our window
    root.geometry('800x600')
    #Add a title to our window
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Exercise 10.1

(U) We have a lovely grid with labels. We don't want to disturb our grid, so let's put a new `Frame` on top. Notice how our frame lines up on the grid. It makes it really easy to see how setting the `row` and `column` alignment works and also the `rowspan` and `columnspan`.

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #-----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #-----------------------------------

        #--- Add Frame 1 ---
        self.add_frame1()
        #--------------------


    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------

    def colorgen(self):
```

```
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)


def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Exercise 10.2

Great! Now let's put a widget in our frame. Notice that while you can't mix `grid` and `pack` inside a conatainer, we can use `pack` inside our `Frame` even though we were using `grid` for our top level window.

Exercise for reader: our Frame was red, but now that we added a text widget it doesn't look red anymore. What happened!? If we actually wanted a red text widget, what should we do differently?

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #---------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #---------------------------------

        #----Add Frame 1-------
        self.add_frame1()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------

    def colorgen(self):
        '''
        Generator function that alternates between red and blue
```

```
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        A frame is a nice way to show how to map out a grid
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)


def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Exercise 10.3

Let's add some another frame.

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #-----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #-----------------------------------

        #----Add Frame 1-------
        self.add_frame1()

        #---- Add Frame 2 -----
        self.add_frame2()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #------------------------------------------------------------
```

```
    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)

    def add_frame2(self):
        # Green frame!
        self.frame2 = Frame(self,bg='green')
        self.frame2.grid(row=0,column=0,rowspan=5,columnspan=6,sticky=self.ALL)



def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

What if we wanted our GUI to do something when we click on our green frame? `Frame` doesn't allow us to set `command`. But wait! Hope is not lost! We can bind our frame to an event...

Binding to an event

(U) Not all widgets have a `command` option, but that doesn't mean you can't interact with them. Also, there may be instances when you want a response from the user other than a mouse click (which is what the built in `command` function responds to). In these instances, you want to bind your widget to the appropriate event.

Format: modifier (optional) - event type - detail (optional)
For example: `<Button-1>` modifier is `none`, event type is `Button` and detail is one. This means the event is the left mouse button was clicked. For the right mouse button, you would use two as your detail.

Common event types: `Button`, `ButtonRelease`, `KeyRelease`, `Keypress`, `FocusIn`, `FocusOut`, `Leave` (when the mouse leaves the widget), and `MouseWheel`.

Common modifiers: `Alt`, `Any` (used like `<Any-KeyPress>`), `Control`, `Double` (used like `<Double-Button-1>`)

Common details: These will vary widely based on the event type. Most commonly you will specify the key for `KeyPress`, ex: `<KeyPress-F1>`

Exercise 10.4

Let's bind our green frame to the left mouse button click and print out the coordinates of the click. Since printing to our notebook or the command line is not terribly useful for GUI's let's also display the coordinates in our text field

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #----------------------------------

        #----Add Frame 1-------
        self.add_frame1()

        #---- Add Buttons -----
        self.add_frame2()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #------------------------------------------------------------
```

```
        def colorgen(self):
            '''
            Generator function that alternates between red and blue
            '''
            while True:
                yield 'red'
                yield 'blue'

        def checkers_main_win(self):
            '''
            Creates a checkerboard pattern grid layout
            '''
            colors = self.colorgen()

            for r in range(self.mainwin_rows):
                for c in range(self.mainwin_cols):
                    txt = 'Item {0}, {1}'.format(r,c)
                    l = Label(self,text=txt,bg=next(colors))
                    l.grid(row=r,column=c,sticky=self.ALL)


        def add_frame1(self):
            '''
            Add a frame with a text area to put stuff in.
            '''

            self.frame1=Frame(self,bg='red')
            self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
            self.frame1.text_w=Text(self.frame1)
            self.frame1.text_w.pack(expand=True,fill=BOTH)

        def add_frame2(self):
            # Green frame!
            self.frame2 = Frame(self,bg='green')
            self.frame2.grid(row=0,column=0,rowspan=5,columnspan=6,sticky=self.ALL)
            self.frame2.bind('<Button-1>',self.frame2_handler)

        def frame2_handler(self, event):
            '''Handles events from frame2'''
            msg = 'Frame 2 clicked at {} {}'.format(event.x,event.y)
            print(msg)
            self.frame1.text_w.delete(1.0,END)
            self.frame1.text_w.insert(END, msg)

    def main():
        root=Tk()
        root.geometry('800x600')
        root.title('Awesome Gui -- It is way COOL')
        app = Application(master=root)

        app.mainloop()
```

```
if __name__ == '__main__':
    main()
```

(U) Example 10.5

Great! Let's add some buttons now. Even though buttons have a `command` attribute, it can be tricky to pass information about the button being clicked using it. We would have to write a separate function for each button! Instead, let's use a key binding so we can access the button `text` from the event. Notice when we arrange the buttons we have to allign them by incriments of three since they span three columns.

Doc ID: 6689695

```
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #---------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #---------------------------------

        #----Add Frame 1-------
        self.add_frame1()

        #----Add Frame 2-------
        self.add_frame2()
        #---- Add Buttons -----
        self.add_buttons()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
```

```
        #-------------------------------------------------------------

    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)

    def add_frame2(self):
        # Green frame!
        self.frame2 = Frame(self,bg='green')
        self.frame2.grid(row=0,column=0,rowspan=5,columnspan=6,sticky=self.ALL)
        self.frame2.bind('<Button-1>',self.frame2_handler)

    def frame2_handler(self, event):
        '''Handles events from frame2'''
        msg = 'Frame 2 clicked at {} {}'.format(event.x,event.y)
        print(msg)
        self.frame1.text_w.delete(1.0,END)
        self.frame1.text_w.insert(END, msg)


    def add_buttons(self):
        ''' Add buttons to the bottom'''
        self.button_list=[]
        button_labels = ['Red','Blue','Green','Black','yellow']
```

```
            for c,bt in enumerate(button_labels):
                b = Button(self,text=bt)
                #span three columns and set the column alignment as
                #multiples of three.
                b.grid(row=10,column=c*3,columnspan=3,sticky=self.ALL)
                #Bind buttons to button click.
                b.bind('<Button-1>',self.buttons_handler)
                self.button_list.append(b)

        def buttons_handler(self,event):
            '''
            Event Handler for the buttons
            '''
            button_clicked = event.widget['text']
            print(button_clicked)

            self.frame1.text_w.delete(1.0,END)
            self.frame1.text_w.insert(END, button_clicked)

    def main():
        root=Tk()
        root.geometry('800x600')
        root.title('Awesome Gui -- It is way COOL')
        app = Application(master=root)

        app.mainloop()

    if __name__ == '__main__':
        main()
```

(U) Dialogs

(U) `Tkinter` provides a collection of ready-made dialog boxes to use:

- `showinfo`
- `showwarning`
- `showerror`
- `askquestion`
- `askokcancel`
- `askyesno`
- `askyesnocancel`
- `askretry_cancel`