CS 1
Introduction to Computer Programming

Lecture 12: October 28, 2015

More Graphics and Event Handling

Last time

- Testing
- Graphics
  - Introductory concepts
    - GUIs, pixels, pixel coordinates
  - Python graphics with **`Tkinter`**

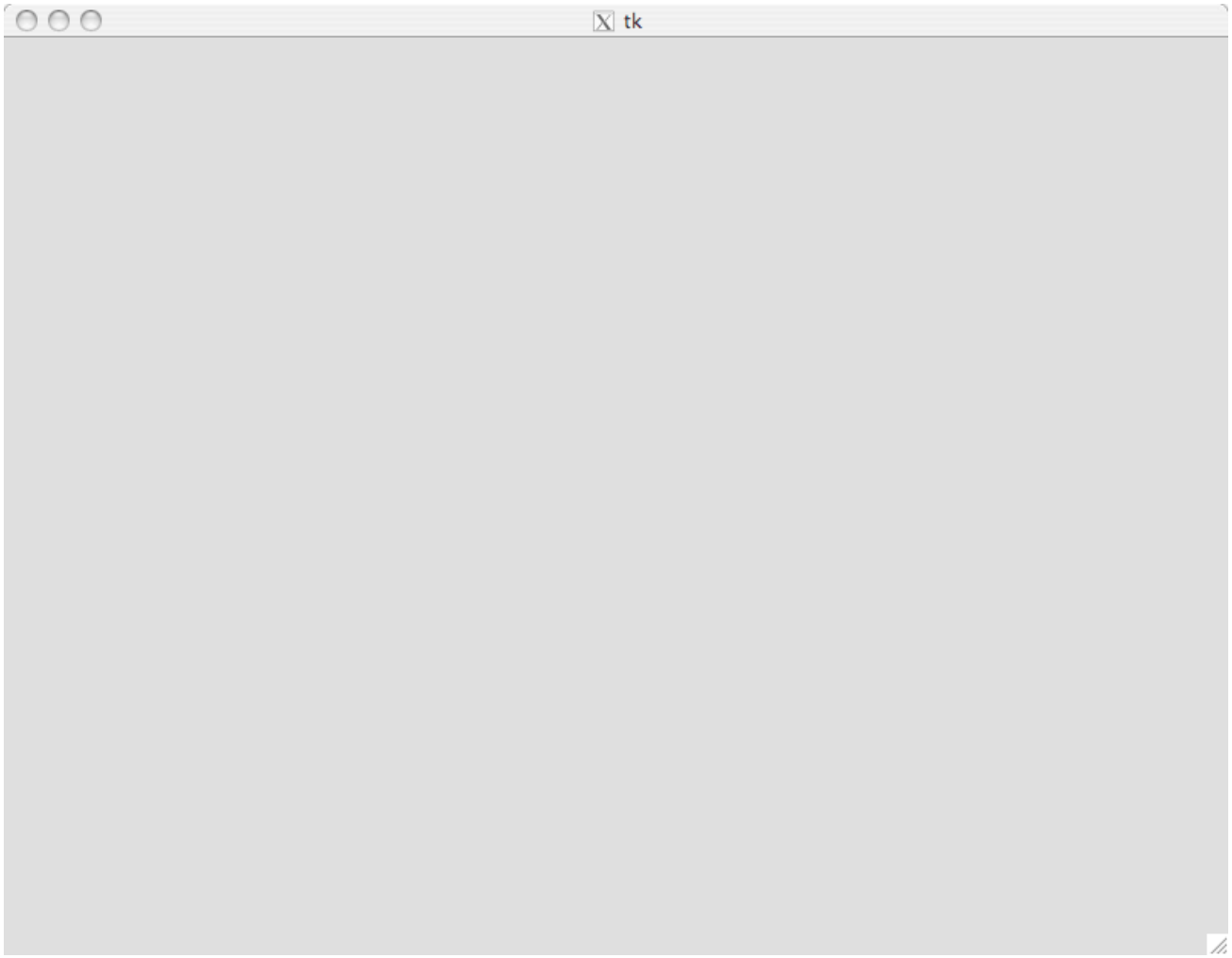
Today

- More graphics
  - Additional graphics elements
  - Event handling
  - Event loops

Previous (simple) example

```
# File: tkinter1.py
from Tkinter import *
root = Tk()
root.geometry('800x600')
raw_input('Press <return> to quit.')
```

Result of the simple example



Result of the simple example

- Much as we love blank windows, we want to do more than this!
- The root window is basically just a container in which we can put other things
- We will put a drawing surface called a canvas inside it
  - canvas: analogy to painter's canvas

Adding a canvas

- Let's add two new lines to the example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
raw_input('Press <return> to quit.')
```

Adding a canvas

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- This creates a new canvas object called **`c`**
- Its *parent* is the **`root`** object
  - it will be located entirely inside that object on screen
- Its dimensions will be 800x600 pixels
  - note keyword arguments: **`width`**, **`height`**

Adding a canvas

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- A canvas is a Python object too, so it has methods
- The **`pack`** method positions the canvas inside its parent (the **`root`** object)
- Since they are both the same size, the canvas completely covers the **`root`** object

Adding a canvas

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- Without this line, the canvas will never show up on the screen!
  - So don't leave it out!

Drawing

- Now we've created
  - the root window
  - the canvas
- It's time to do some actual drawing on the canvas

Drawing

```
# ... as before ...
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
                       fill='red', outline='red')
```

- This creates a rectangle `r` on the canvas `c`
- `create_rectangle` is a method of the canvas object `c`

Drawing

```
# ... as before ...
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
            fill='red', outline='red')
```

- The first four arguments: `0`, `0`, `50`, `50` mean:
  - rectangle's upper left-hand corner is at location (0, 0)
  - rectangle's lower right-hand corner is at location (50, 50)
  - so it's actually a square of dimensions 50x50 pixels

Drawing

```
# ... as before ...
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
                       fill='red', outline='red')
```

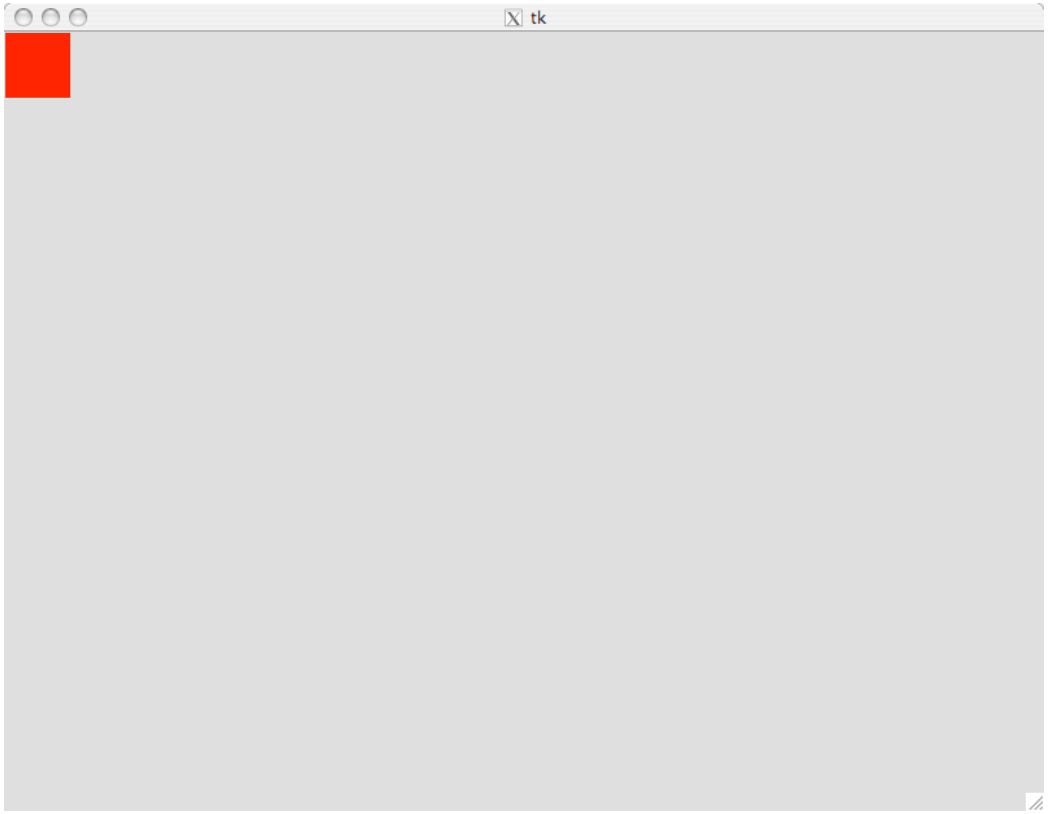
- The `fill` is the color inside the square
  - set it to be `'red'` because we like red!
- The outline is the color of the edges of the square
  - set to be `'red'` to make the entire square red

Drawing

```
# ... as before ...
r = c.create_rectangle(0, 0, 50, 50, \
          fill='red', outline='red')
raw_input('Press <return> to quit.')
```

- The **`raw_input`** line again makes the image visible until we hit the return key on the terminal
- This is a *very* crude way of interacting with a graphical program!
  - We'll see better ways soon

Result



- This is boring
- Let's add more stuff!

Drawing more

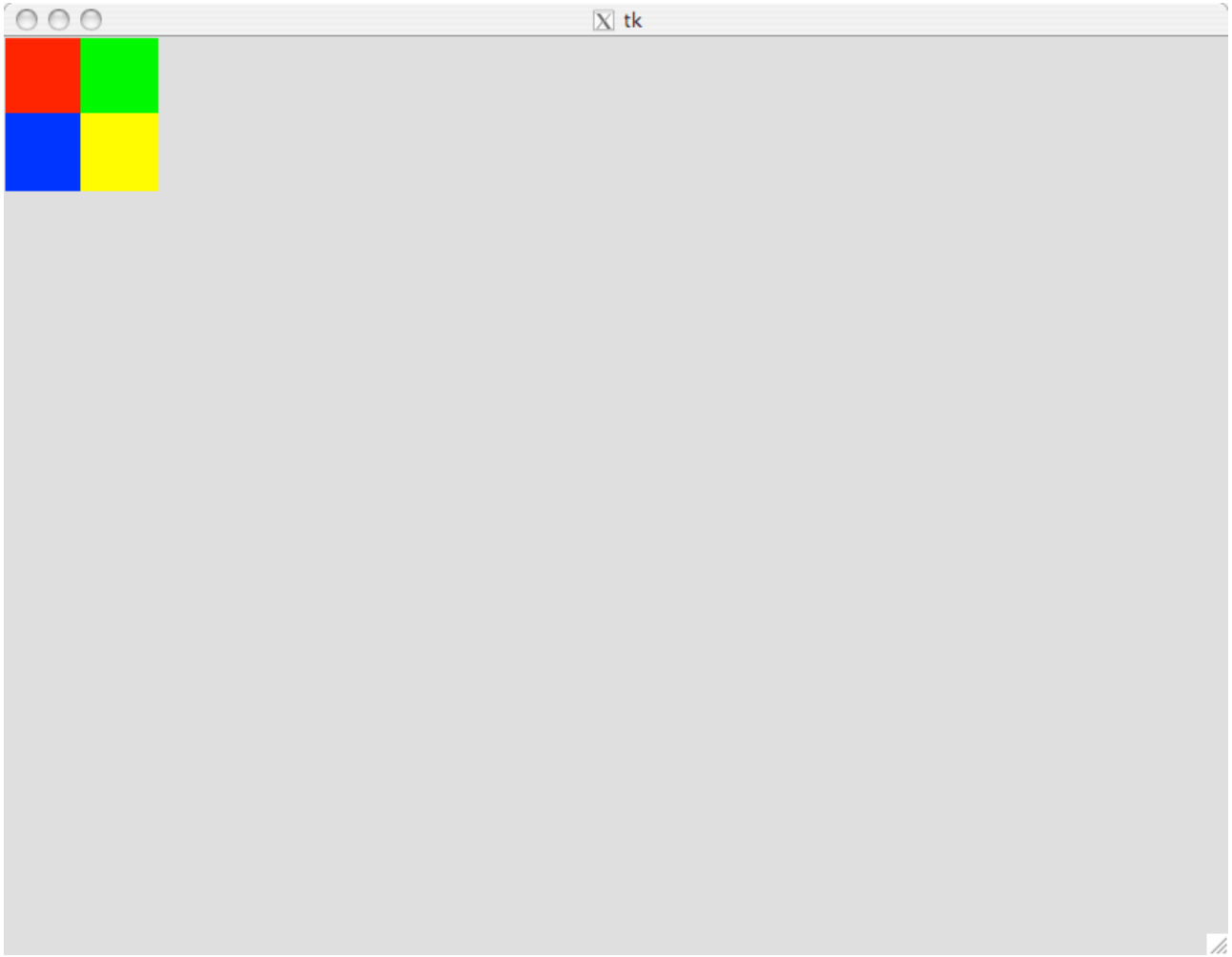
```
# ... as before ...
r = c.create_rectangle(0, 0, 50, 50, \
          fill='red', outline='red')
# ... add extra lines here ...
raw_input('Press <return> to quit.')
```

- We'll put extra lines between the **`create_rectangle`** line and the **`raw_input`** line
  - won't re-type those lines to save space on slides

Drawing more

```
# ... previous stuff ...
r2 = c.create_rectangle(0, 50, 50, 100, \
  fill='blue', outline='blue')
r3 = c.create_rectangle(50, 0, 100, 50, \
   fill='green', outline='green')
r4 = c.create_rectangle(50, 50, 100, 100,\
   fill='yellow', outline='yellow')
raw_input('Press <return> to quit.')
```

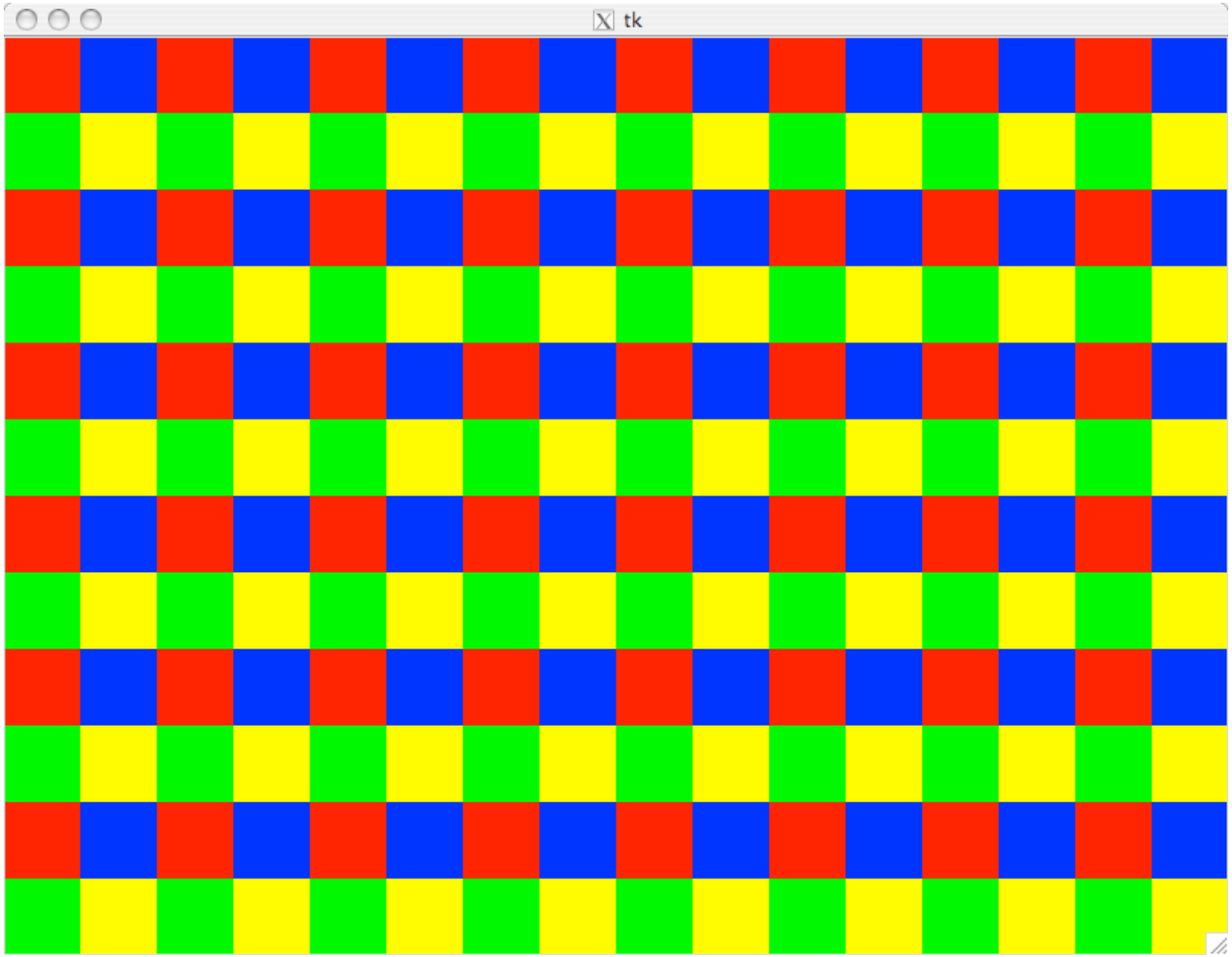
Result



Drawing more

- With only a few simple additions, we can generate more complicated images...

Drawing more



Moving on

- Now we know how to draw colored squares
- We will now create some different graphical objects

Starting point

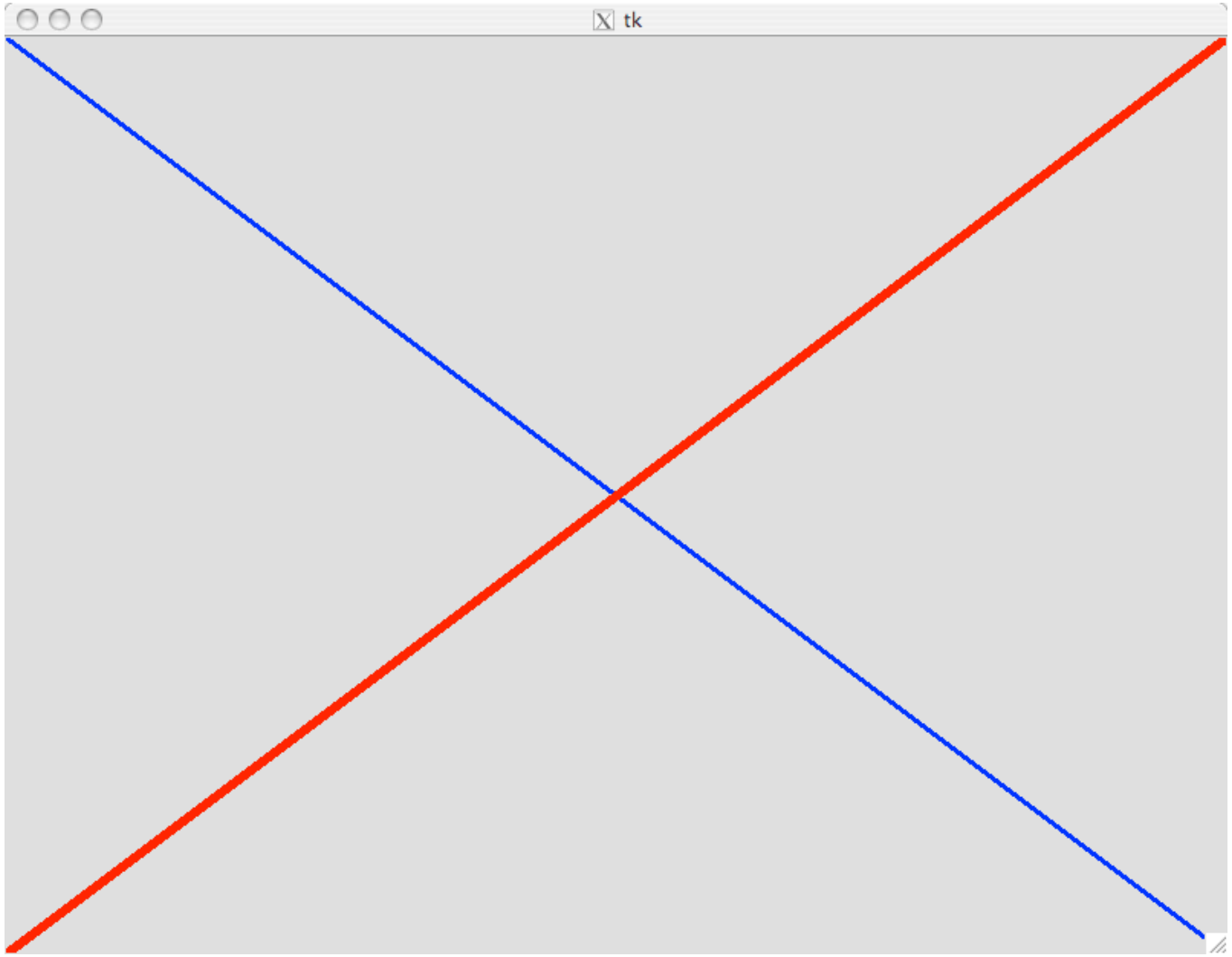
```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- This part of the code will stay the same

Lines

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
line1 = c.create_line(0, 0, 800, 600,
        fill='blue', width=3)
line2 = c.create_line(800, 0, 0, 600,
        fill='red', width=6)
raw_input('Press <return> to quit')
```

Result



create_line

- `create_line` is a method on canvas objects
- It creates one or more connected lines with particular properties
- Arguments:

```
create_line(x1, y1, x2, y2, ...,
  option1=value1, ...)
```

create_line

```
create_line(x1, y1, x2, y2, ...,
  option1=value1, ...)
```

- **x1** and **y1** are the initial point (where the line begins)

create_line

```
create_line(x1, y1, x2, y2, ...,
  option1=value1, ...)
```

- **x2** and **y2** are the next point
- A line is drawn on the canvas between coordinates **(x1, y1)** and coordinates **(x2, y2)**

create_line

```
create_line(x1, y1, x2, y2, x3,
  y3, ..., option1=value1, ...)
```

- There may or may not be more points
- Here, `x3` and `y3` are the next point
- A line is drawn on the canvas between coordinates `(x2, y2)` and coordinates `(x3, y3)`
- A single `create_line` function call can create a series of connected lines

Python notes

- `create_line` is an unusual function
- It can take an *arbitrary* number of arguments
  - the `x1`, `y1`, `x2`, `y2`, `x3`, `y3` etc. arguments
  - (We haven't seen how to do this yet!)
  - Must have at least `x1`, `y1`, `x2`, `y2` arguments (this makes one line)
  - If any more `x`, `y` pairs, they define the endpoints of subsequent lines

Python notes

- `create_line` can also take an arbitrary number of *keyword arguments*
  - the `option=value` arguments
  - (We saw how to do this last time)

- Examples:
  - `fill='blue'` (color of the line)
  - `width=3` (width of the line in pixels)

Back to the example

```
line1 = c.create_line(0, 0, 800, 600,
        fill='blue', width=3)
```

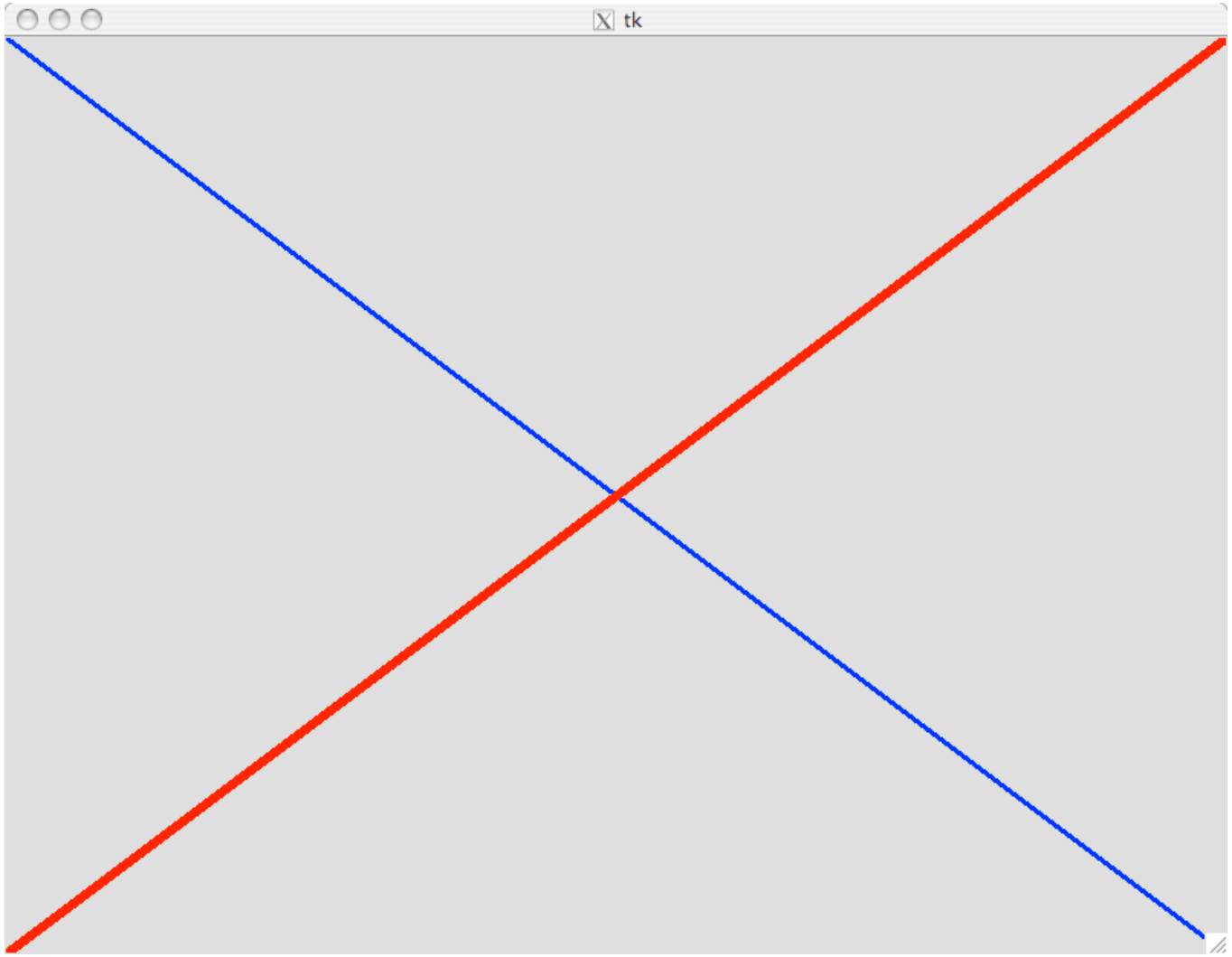
- This means:
  - create a line between coordinates `(0, 0)` and `(800, 600)`
  - (the upper-left corner and the lower-right corner)
  - This line should be `blue`
  - The width of the line should be `3` pixels
  - The resulting line should be named `line1`

Back to the example

```
line2 = c.create_line(800, 0, 0, 600,
        fill='red', width=6)
```

- This means:
  - create a line between coordinates `(800, 0)` and `(0, 600)`
  - (the upper-right corner and the lower-left corner)
  - This line should be `red`
  - The width of the line should be `6` pixels
  - The resulting line should be named `line2`

Result, again



Drawing more lines

- Let's try different line drawing commands:

```
line1 = c.create_line(100, 100, 400, 100,
                      100, 400, 400, 400,
                      fill='blue',
                      width=3)
```

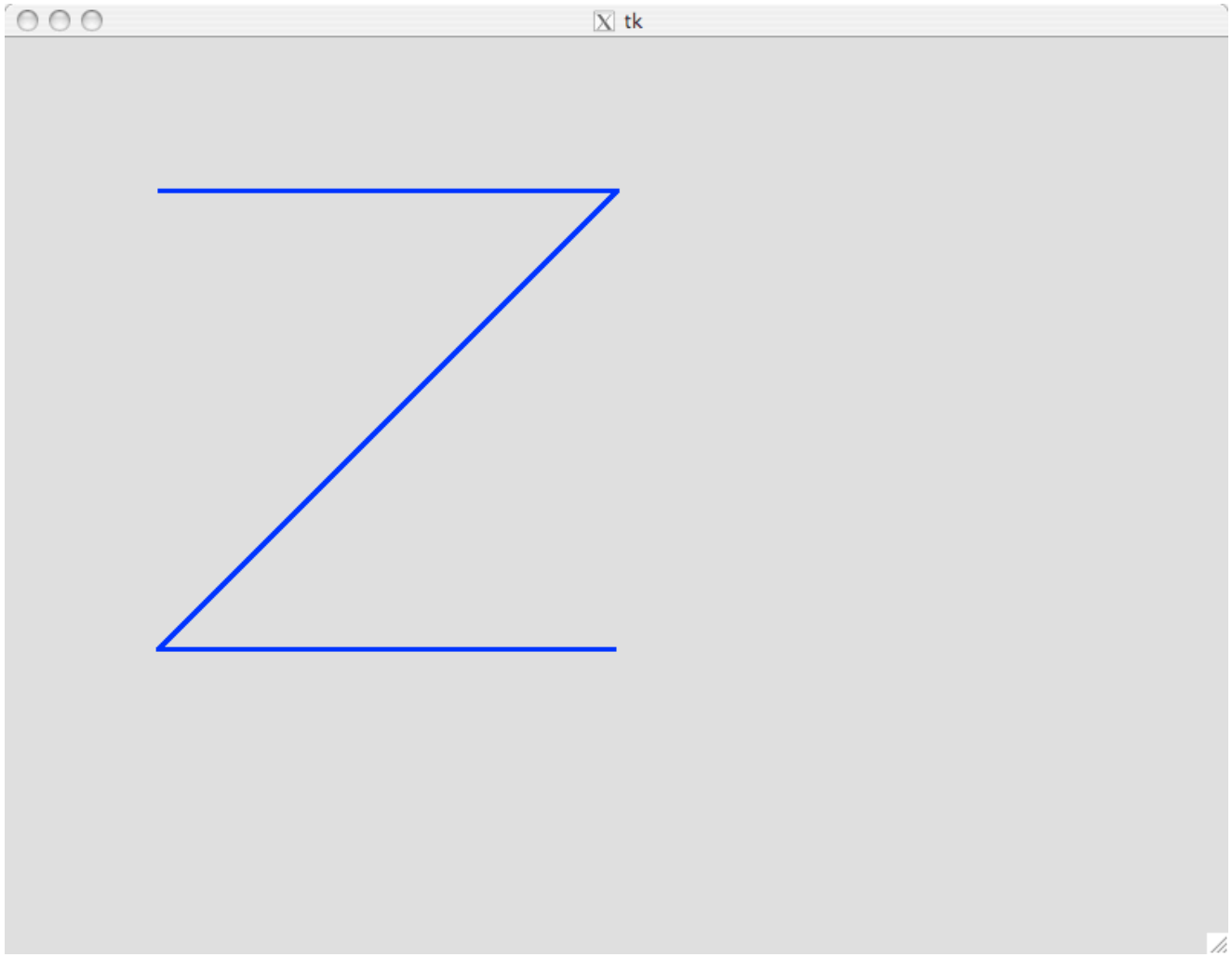
- This command creates 3 lines joined end-to-end
- All colored blue with a width of 3 pixels

Drawing more lines

```
line1 = c.create_line(100, 100, 400, 100,
                      100, 400, 400, 400,
                      fill='blue',
                      width=3)
```

- First line: `(100, 100)` to `(400, 100)` (horizontal)
- Second line: `(400, 100)` to `(100, 400)` (diagonal)
- Third line: `(100, 400)` to `(400, 400)` (horizontal)
- Gives a zig-zag 'Z' pattern

Result



Beyond lines

- Lines are only one of many things we can draw on Tkinter canvases
  - (Saw rectangular boxes earlier)
- Many other things can be drawn on canvases
  - polygons, arcs, text, etc.
- Commands are similar to what we've already seen
- For fun, we'll look at one more example: ovals

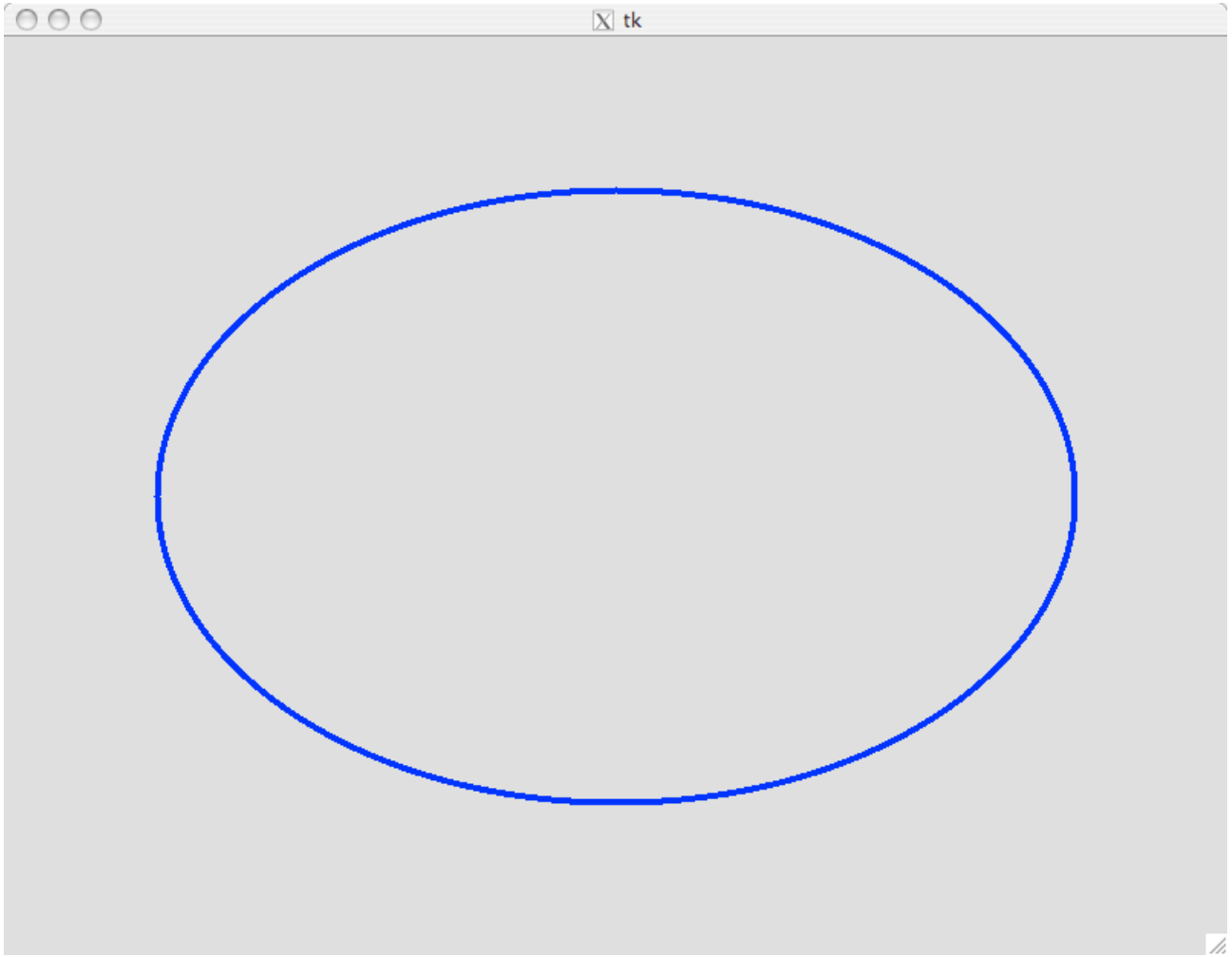
Ovals

- An oval is an elliptical shape which fits neatly inside a rectangle
  - edges touch the outer edges of the rectangle
- If the rectangle is a square, the corresponding oval is a circle
- Commands to draw ovals in `Tkinter` are very similar to rectangle-drawing commands

Oval example 1

```
oval = c.create_oval(100, 100, 700, 500,
                     outline='blue',
                     width=4)
```

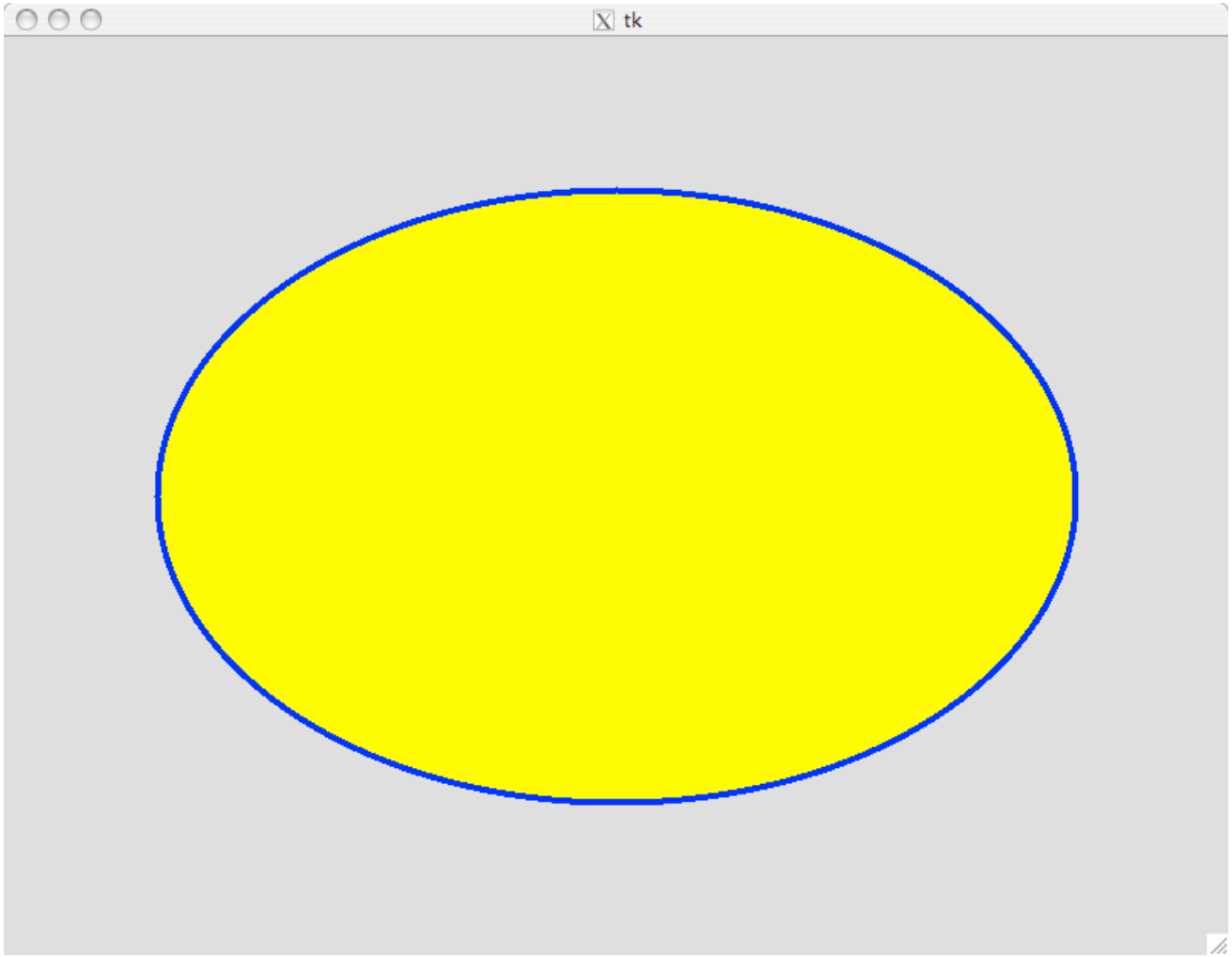
Oval example 1



Oval example 2

```
oval = c.create_oval(100, 100, 700, 500,
                     outline='blue',
                     fill='yellow',
                     width=4)
```

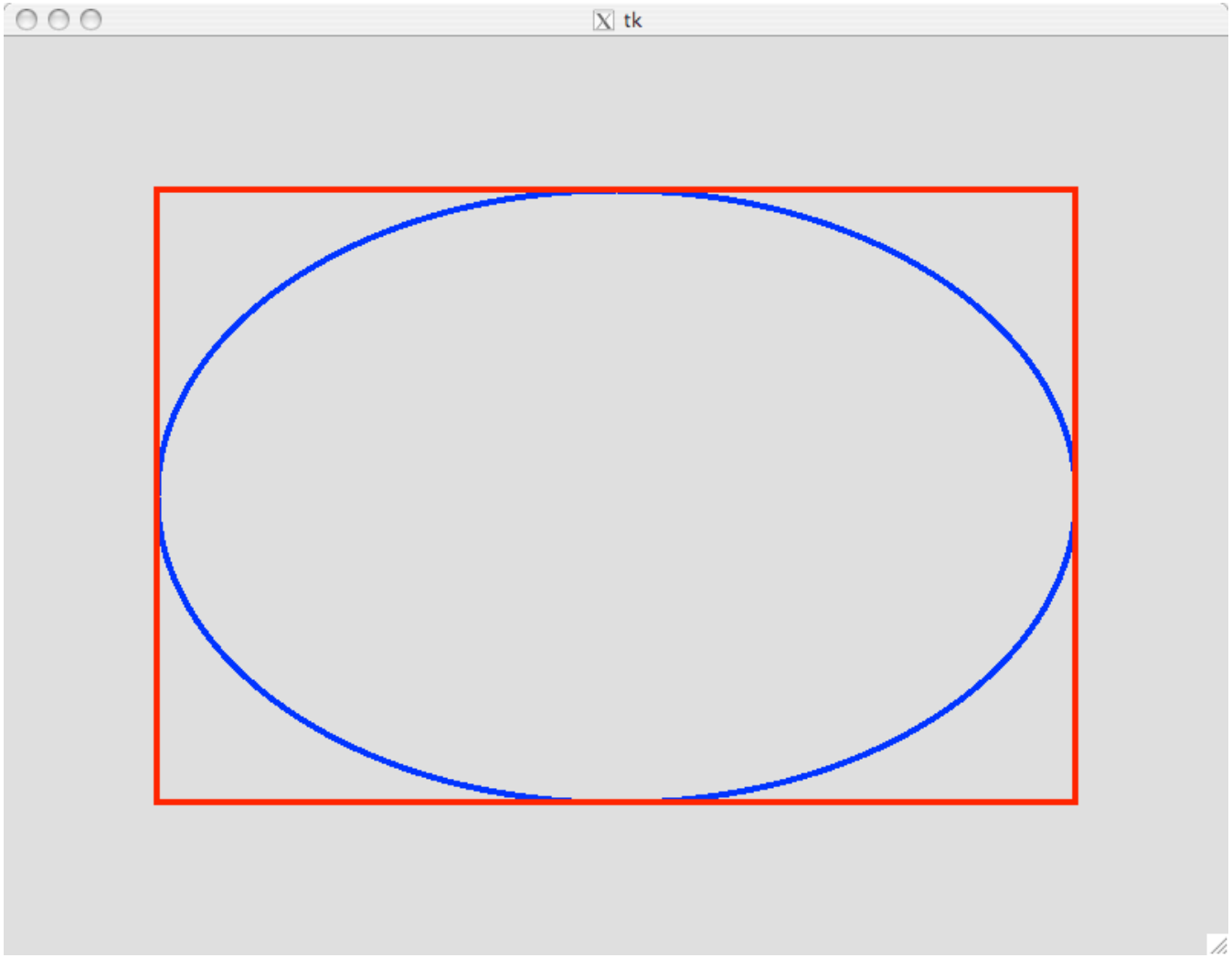
Oval example 2



Oval example 3

```
oval = c.create_oval(100, 100, 700, 500,
                     outline='blue',
                     width=4)
rect = c.create_rectangle(100, 100,
                     700, 500,
                     outline='red',
                     width=4)
```

Oval example 3

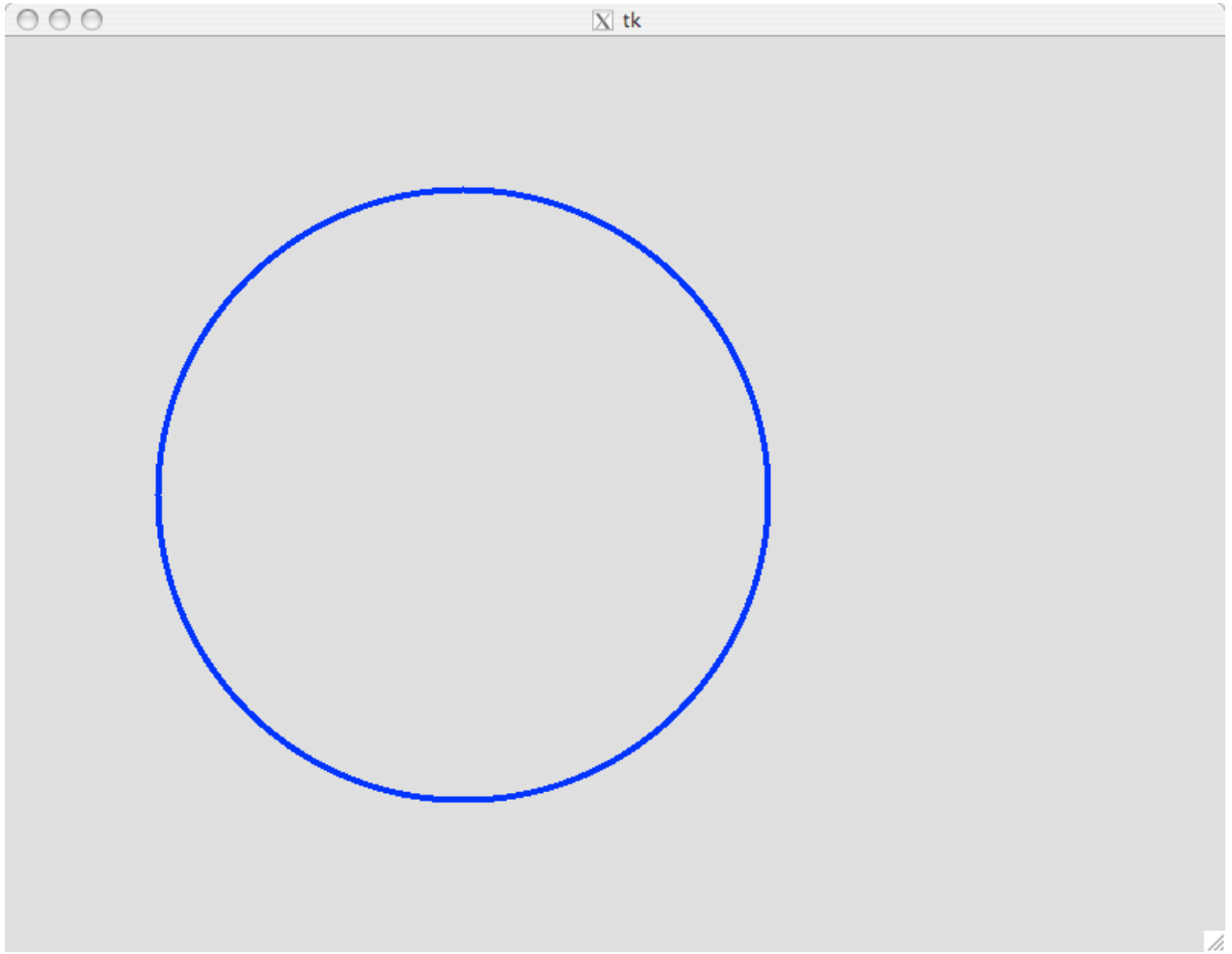


Oval example 4

```
oval = c.create_oval(100, 100, 500, 500,
                     outline='blue',
                     width=4)
```

- (This is a circle)

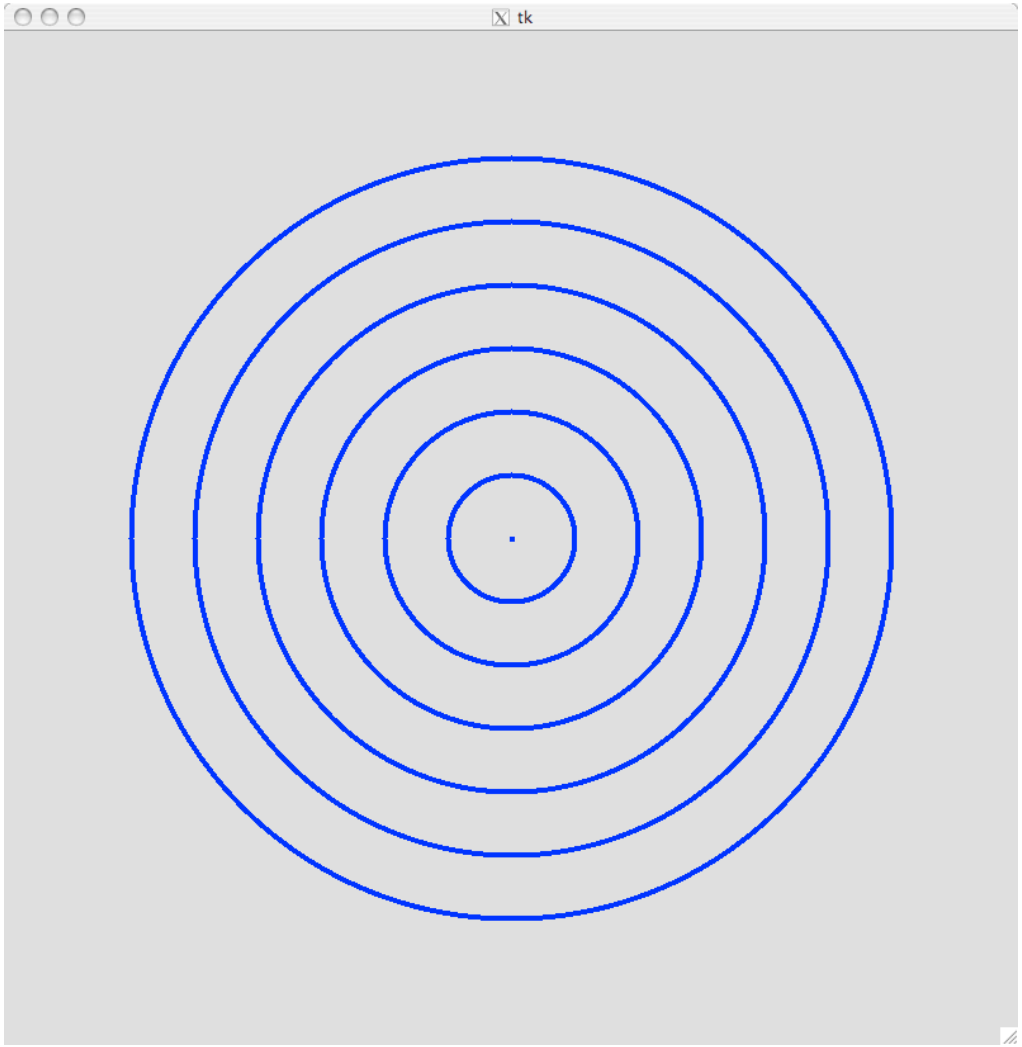
Oval example 4



Oval example 5

- It's easy to put graphics commands inside loops to create interesting images

Oval example 5



Oval example 5

```
from Tkinter import *
root = Tk()
root.geometry('800x800')
c = Canvas(root, width=800, height=800)
c.pack()
for i in range(50, 400, 50):
    x1 = 50 + i
    y1 = x1
    x2 = 750 - i
    y2 = x2
    c.create_oval(x1, y1, x2, y2,
                  outline='blue', width=4)
raw_input('Press <return> to quit.')
```

New topic!

Event handling

- So far, at the end of all of our graphics programs we wrote:

```
raw_input('Press <return> to quit')
```

- This is just to make sure the canvas stayed up while we looked at the graphics
- This is not the normal way to use **`Tkinter`**
- And in addition...

Event handling

- Our graphics programs have been totally *static*
- They display an image and that's all
- In reality, many more things can be done:
  - mouse click to exit the program
  - mouse click to create/delete/move graphical objects
  - bind keys to actions ('q' might mean 'quit')
- In other words, we want our graphics programs to be more *dynamic*
  - to respond to user input

Events and the event loop

- **`Tkinter`** programs are normally structured around an *event loop*
- The event loop is something that waits and "listens" for *events* happening on a graphical object

Events and the event loop

- Events are something you do while the program is running to notify the program that you want some action to occur
- Events may include:
  - key presses
  - mouse clicks
  - mouse movement
  - etc.

Events and the event loop

- Some events you might want:
  - "When I press the 'q' key, I want the program to exit."
  - "When I click the mouse, I want a square to be drawn on the canvas at the location of the mouse cursor."
  - "When I click the mouse, I want all the squares on the screen to be removed from the screen."

Specifying events

- Specifying events requires that you do these things:

1. Decide which graphical object is going to handle the events
   - *e.g.* the canvas, the root window
2. Decide which action will trigger which event
   - *e.g.* a mouse click, a key press
3. Write a function to handle each event
4. *Bind* the action to the event

mainloop

- Before we get into the details of this, we introduce the command which starts the event loop
- Assuming the root window is called `root`, we write

```
root.mainloop()
```

- to start the event loop

mainloop

```
root.mainloop()
```

- is an unusual method call
- Unlike most method calls, it doesn't normally return!
  - just loops forever
- Normally, it's a bad thing if a function or method call never terminates
- Here, it's what you want

mainloop

- What we need to know:
  - The event loop starts up when `root.mainloop()` executes
  - When events happen, the event loop will catch them and *dispatch* them to the functions that handle them
  - The details of how this works aren't important to us right now

mainloop

- Let's return to our first example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
                       fill='red', outline='red')
raw_input("Press <return> to quit")
```

mainloop

- Let's return to our first example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
                       fill='red', outline='red')
raw_input("Press <return> to quit")
```

mainloop

- Let's return to our first example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
                       fill='red', outline='red')
root.mainloop()
```

mainloop

- We added the

```
root.mainloop()
```

- line in place of the `raw_input` line
- The drawing doesn't change
- Now, the only way to exit the program is to close the window or to quit Python
- So far, haven't added any code to handle any events

Event 1: q for quit

- Let's add an event so that pressing the `q` key on the canvas quits the application
- We will add just one line before the **`mainloop`** line:

```
root.bind('<q>', quit)
```

- Let's look at this line in detail

Event 1: q for quit

```
root.bind('<q>', quit)
```

- This says:
- The graphical object that will handle the event is the `root` object
- The event to be handled is when the user presses the `q` key on the keyboard
  - `Tkinter` represents this as the string `'<q>'`
- When the `q` key is pressed, the built-in `quit` function will be executed
  - This causes Python to exit immediately

Event 1: q for quit

```
root.bind('<q>', quit)
```

- We say that this method call *binds* the event (pressing the `q` key on the keyboard) to the function `quit` (which handles the event)

Event 1: q for quit

- This works, but there is one odd thing
- When `q` is pressed, the program exits
  - but before it does, it prints out something like:

```
<Tkinter.Event instance at 0x776058>
```

- What does this mean?

Event 1: q for quit

- The `quit` function usually gets no arguments
- If it gets an integer argument, it passes it to the operating system
  - (the reason isn't important)
- If it gets a non-integer argument, it prints it and then exits
- Somehow, here it got an argument of type `Tkinter.Event`

Event 1: q for quit

- What happened:
- When the `q` key on the keyboard was pressed
  1. `Tkinter` created a `Tkinter.Event` object representing the key press event
  2. `Tkinter`'s event loop (`mainloop`) checked to see if the `q` key was bound to any function
  3. It found that `q` was bound to the `quit` function
  4. It called `quit` with the `Tkinter.Event` object as its argument
  5. `quit` printed the event and then made Python exit

Event 1: q for quit

- Another weird thing...
- Look at the line:

```
root.bind('<q>', quit)
```

- `quit` is the name of a Python function
- The bind method of the root object can take a *function* as an argument!
- Python allows functions to be treated as data
  - functions are "first-class"
- We will talk about this again later in the course

Summary

- We've used the following **`Tkinter`** features:
  - **`Tk()`** function to create the root window
    - **`geometry()`** method
    - **`mainloop()`** method
    - **`bind()`** method
  - **`Canvas()`** function to create the canvas object
    - **`pack()`** method
    - **`create_rectangle()`** method
    - **`create_line()`** method
    - **`create_oval()`** method

Monday

- More on event handling
  - more events
  - callback functions
  - What's in an event?
- Talk about graphical object *handles*
  - a way to manipulate graphical objects that have been created