CS 115 Lecture 6

Graphics

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

17 September 2015

The `graphics` library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?

The `graphics` library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?
- The `graphics` library by John Zelle is one way.
  - Not part of Python: a **third-party** library.

The `graphics` library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?
- The `graphics` library by John Zelle is one way.
  - Not part of Python: a **third-party** library.
  - Download it from the 115 web page, or Zelle's site.
  - Then what?

The `graphics` library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?
- The `graphics` library by John Zelle is one way.
  - Not part of Python: a **third-party** library.
  - Download it from the 115 web page, or Zelle's site.
  - Then what?
  - Either: put it in the same directory as your code...
  - ...or find your system Python directory and put it there.

The graphics library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?
- The `graphics` library by John Zelle is one way.
  - Not part of Python: a **third-party** library.
  - Download it from the 115 web page, or Zelle's site.
  - Then what?
  - Either: put it in the same directory as your code...
  - ...or find your system Python directory and put it there.
    - ★ `http://www.cs.uky.edu/~keen/115/graphics-fix.html`
    - ★
      ```
      import sys
      print(sys.path)
      ```
      and find `site-packages`

The graphics library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?
- The `graphics` library by John Zelle is one way.
  - Not part of Python: a **third-party** library.
  - Download it from the 115 web page, or Zelle's site.
  - Then what?
  - Either: put it in the same directory as your code...
  - ...or find your system Python directory and put it there.
    - ★ `http://www.cs.uky.edu/~keen/115/graphics-fix.html`
    - ★
      ```
      import sys
      print(sys.path)
      ```
      and find `site-packages`
- Other graphics packages take different approaches:
  - `turtle` does **turtle graphics**, based around moving a cursor.
  - `Tkinter` does graphical user interfaces based around **widgets** like checkboxes, labels, text fields, ...

The graphics library

So far all our programs have interacted with the user through standard input and output (the shell window)

- Plain-text input (`input`) and output (`print`).
- Can we do something that looks nicer?
- The `graphics` library by John Zelle is one way.
  - Not part of Python: a **third-party** library.
  - Download it from the 115 web page, or Zelle's site.
  - Then what?
  - Either: put it in the same directory as your code...
  - ...or find your system Python directory and put it there.
    - ★ `http://www.cs.uky.edu/~keen/115/graphics-fix.html`
    - ★
      ```
      import sys
      print(sys.path)
      ```
      and find `site-packages`
- Other graphics packages take different approaches:
  - `turtle` does **turtle graphics**, based around moving a cursor.
  - `Tkinter` does graphical user interfaces based around **widgets** like checkboxes, labels, text fields, ...

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.
- An object is a particular thing of that type.
  - So `int` is a class, and `42` is an object of that class.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.
- An object is a particular thing of that type.
  - So `int` is a class, and `42` is an object of that class.
  - `str` is a class, and `"abracadabra"` an object.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.
- An object is a particular thing of that type.
  - So `int` is a class, and `42` is an object of that class.
  - `str` is a class, and `"abracadabra"` an object.
  - `Point` is a class, and `Point(100, 100)` an object.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.
- An object is a particular thing of that type.
  - So `int` is a class, and `42` is an object of that class.
  - `str` is a class, and `"abracadabra"` an object.
  - `Point` is a class, and `Point(100, 100)` an object.
- Why did we have to write it as `Point(100, 100)`?
  - Unlike Python's built-in types, most classes don't have literals.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.
- An object is a particular thing of that type.
  - So `int` is a class, and `42` is an object of that class.
  - `str` is a class, and `"abracadabra"` an object.
  - `Point` is a class, and `Point(100, 100)` an object.
- Why did we have to write it as `Point(100, 100)`?
  - Unlike Python's built-in types, most classes don't have literals.
  - Instead you call a **constructor** to make an object.
    - ★ A special function that returns a new object.
    - ★ Name of the constructor = name of the class.

Classes, objects and constructors

- Object oriented programming: classes, objects, and methods.
- A **class** is a type (kind of thing that can be stored in a variable)
  - Especially a user- or library-defined type.
  - But in Python, `str`, `float`, etc. are also classes.
- An object is a particular thing of that type.
  - So `int` is a class, and `42` is an object of that class.
  - `str` is a class, and `"abracadabra"` an object.
  - `Point` is a class, and `Point(100, 100)` an object.
- Why did we have to write it as `Point(100, 100)`?
  - Unlike Python's built-in types, most classes don't have literals.
  - Instead you call a **constructor** to make an object.
    - ★ A special function that returns a new object.
    - ★ Name of the constructor = name of the class.

Classes in the `graphics` library

The Zelle `graphics` library defines several classes. Among them:

- `GraphWin` — a window for drawing graphics.
- `Point` — an (x, y) coordinate.
- `Line` — a line segment with two endpoints.
- `Circle` — a circle with center and radius.
- `Rectangle` — a rectangle (given by two opposite corners).
- `Oval` — an oval that fits inside a "bounding box"
- `Polygon` — defined by connecting a sequence of points.
- `Text` — text with a given string, position, size, etc.
- `Entry` — text with a given string, position, size, etc.

Classes in the `graphics` library

The Zelle `graphics` library defines several classes. Among them:

- `GraphWin` — a window for drawing graphics.
- `Point` — an (x, y) coordinate.
- `Line` — a line segment with two endpoints.
- `Circle` — a circle with center and radius.
- `Rectangle` — a rectangle (given by two opposite corners).
- `Oval` — an oval that fits inside a "bounding box"
- `Polygon` — defined by connecting a sequence of points.
- `Text` — text with a given string, position, size, etc.
- `Entry` — text with a given string, position, size, etc.
- The complete reference: `http://mcsp.wartburg.edu/zelle/python/graphics/graphics.html`

Classes in the `graphics` library

The Zelle `graphics` library defines several classes. Among them:

- `GraphWin` — a window for drawing graphics.
- `Point` — an (x, y) coordinate.
- `Line` — a line segment with two endpoints.
- `Circle` — a circle with center and radius.
- `Rectangle` — a rectangle (given by two opposite corners).
- `Oval` — an oval that fits inside a "bounding box"
- `Polygon` — defined by connecting a sequence of points.
- `Text` — text with a given string, position, size, etc.
- `Entry` — text with a given string, position, size, etc.
- The complete reference: `http://mcsp.wartburg.edu/zelle/python/graphics/graphics.html`

Getting started

- Begin by importing the library:

```
import graphics
```

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - Constructor: `graphics.GraphWin(title, width, height)`
    - Or just `graphics.GraphWin()` ("Graphics Window", 200×200).

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - ★ Constructor: `graphics.GraphWin(title, width, height)`
    - ★ Or just `graphics.GraphWin()` ("Graphics Window", 200×200).
  - Call the constructor, save the new object in a variable:

```
window = graphics.GraphWin("115 Program", 600, 400)
```

    - ★ We'll need it later!

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - ★ Constructor: `graphics.GraphWin(title, width, height)`
    - ★ Or just `graphics.GraphWin()` ("Graphics Window", 200×200).
  - Call the constructor, save the new object in a variable:

```
window = graphics.GraphWin("115 Program", 600, 400)
```

    - ★ We'll need it later!
- The window usually closes when the program exits.
  - Keep it open by waiting for a click:

```
win.getMouse()
```

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - ★ Constructor: `graphics.GraphWin(title, width, height)`
    - ★ Or just `graphics.GraphWin()` ("Graphics Window", 200×200).
  - Call the constructor, save the new object in a variable:

```
window = graphics.GraphWin("115 Program", 600, 400)
```

    - ★ We'll need it later!
- The window usually closes when the program exits.
  - Keep it open by waiting for a click:

```
win.getMouse()
```

    - ★ More about `getMouse` later...

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - Constructor: `graphics.GraphWin(title, width, height)`
    - Or just `graphics.GraphWin()` ("Graphics Window", 200×200).
  - Call the constructor, save the new object in a variable:

```
window = graphics.GraphWin("115 Program", 600, 400)
```

    - We'll need it later!
- The window usually closes when the program exits.
  - Keep it open by waiting for a click:

```
win.getMouse()
```

    - More about `getMouse` later...
  - Close on exit doesn't always work in the IDE
    - ...or when the program crashes.
    - Can eat up lots of system resources and even require a reboot!

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - Constructor: `graphics.GraphWin(title, width, height)`
    - Or just `graphics.GraphWin()` ("Graphics Window", 200×200).
  - Call the constructor, save the new object in a variable:

```
window = graphics.GraphWin("115 Program", 600, 400)
```

    - We'll need it later!
- The window usually closes when the program exits.
  - Keep it open by waiting for a click:

```
win.getMouse()
```

    - More about `getMouse` later...
  - Close on exit doesn't always work in the IDE
    - ...or when the program crashes.
    - Can eat up lots of system resources and even require a reboot!
  - Be safe by calling `win.close()`

Getting started

- Begin by importing the library:

```
import graphics
```

  - Or: `from graphics import *`
- Now we need to create a window to draw in.
  - The class for windows is called `GraphWin`.
    - Constructor: `graphics.GraphWin(title, width, height)`
    - Or just `graphics.GraphWin()` ("Graphics Window", 200×200).
  - Call the constructor, save the new object in a variable:

```
window = graphics.GraphWin("115 Program", 600, 400)
```

    - We'll need it later!
- The window usually closes when the program exits.
  - Keep it open by waiting for a click:

```
win.getMouse()
```

    - More about `getMouse` later...
  - Close on exit doesn't always work in the IDE
    - ...or when the program crashes.
    - Can eat up lots of system resources and even require a reboot!
  - Be safe by calling `win.close()`

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"?

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"? Another class!
  - Constructor: `graphics.Point(x, y)` (`x` and `y` are floats)

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"? Another class!
  - Constructor: `graphics.Point(x, y)` (`x` and `y` are floats)
    - ★ By default, $(0, 0)$ is the upper left.
    - ★ Upside-down compared to math!

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"? Another class!
  - Constructor: `graphics.Point(x, y)` (`x` and `y` are floats)
    - ★ By default, $(0, 0)$ is the upper left.
    - ★ Upside-down compared to math!
  - Can use a constructor (any expression) as an argument:

```
from graphics import Line, Point
diagonal = Line(Point(0, 0), Point(600, 400))
```

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"? Another class!
  - Constructor: `graphics.Point(x, y)` (`x` and `y` are floats)
    - ★ By default, $(0, 0)$ is the upper left.
    - ★ Upside-down compared to math!
  - Can use a constructor (any expression) as an argument:

```
from graphics import Line, Point
diagonal = Line(Point(0, 0), Point(600, 400))
```

- Making the line doesn't actually draw it!
  - One more step: tell it to draw in the window:

```
diagonal.draw(window)
```

  - Why?

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"? Another class!
  - Constructor: `graphics.Point(x, y)` (`x` and `y` are floats)
    - ★ By default, $(0, 0)$ is the upper left.
    - ★ Upside-down compared to math!
  - Can use a constructor (any expression) as an argument:

```
from graphics import Line, Point
diagonal = Line(Point(0, 0), Point(600, 400))
```

- Making the line doesn't actually draw it!
  - One more step: tell it to draw in the window:

```
diagonal.draw(window)
```

  - Why? Programs can have multiple windows!
  - . . . or you might want to set the color first.

Drawing graphics objects

Let's make a line going from the upper left to lower right.
To do that, we use the `Line` class.

- Constructor: `graphics.Line(point1, point2)`
- What's a "point"? Another class!
  - Constructor: `graphics.Point(x, y)` (`x` and `y` are floats)
    - ★ By default, $(0, 0)$ is the upper left.
    - ★ Upside-down compared to math!
  - Can use a constructor (any expression) as an argument:

```
from graphics import Line, Point
diagonal = Line(Point(0, 0), Point(600, 400))
```

- Making the line doesn't actually draw it!
  - One more step: tell it to draw in the window:

```
diagonal.draw(window)
```

  - Why? Programs can have multiple windows!
  - . . . or you might want to set the color first.

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.
  - "Something the object can do."
  - In OOP, methods are how our program interacts with objects.

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.
  - "Something the object can do."
  - In OOP, methods are how our program interacts with objects.
- Syntax: `obj.method(arguments)`
  - `obj` is an object (usually a variable).
  - `method` is the name of the method.

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.
  - "Something the object can do."
  - In OOP, methods are how our program interacts with objects.
- Syntax: `obj.method(arguments)`
  - `obj` is an object (usually a variable).
  - `method` is the name of the method.
- Semantics: Calls the function "method" in obj's class, sending it obj as the object to work on.

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.
  - "Something the object can do."
  - In OOP, methods are how our program interacts with objects.
- Syntax: `obj.method(arguments)`
  - `obj` is an object (usually a variable).
  - `method` is the name of the method.
- Semantics: Calls the function "method" in obj's class, sending it obj as the object to work on.
- Methods can return values just like ordinary functions:

```
x = point.getX()
```

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.
  - "Something the object can do."
  - In OOP, methods are how our program interacts with objects.
- Syntax: `obj.method(arguments)`
  - `obj` is an object (usually a variable).
  - `method` is the name of the method.
- Semantics: Calls the function "method" in obj's class, sending it obj as the object to work on.
- Methods can return values just like ordinary functions:

```
x = point.getX()
```

- The `draw` method does not return anything (like `print`).

```
diagonal.draw(win)
```

Methods

What's up with that `diagonal.draw(window)`?

- `draw` is a **method** of the `Line` class.
  - A method is like a function that works on an object.
  - "Something the object can do."
  - In OOP, methods are how our program interacts with objects.
- Syntax: `obj.method(arguments)`
  - `obj` is an object (usually a variable).
  - `method` is the name of the method.
- Semantics: Calls the function "method" in obj's class, sending it obj as the object to work on.
- Methods can return values just like ordinary functions:

```
x = point.getX()
```

- The `draw` method does not return anything (like `print`).

```
diagonal.draw(win)
```

More shapes: circles

- The `Circle` class represents a circle (unsurprisingly)
- What information is needed to draw a circle?

More shapes: circles

- The `Circle` class represents a circle (unsurprisingly)
- What information is needed to draw a circle?
  - Center – a `Point`.
  - Radius – a number (distance from center to edge).

More shapes: circles

- The `Circle` class represents a circle (unsurprisingly)
- What information is needed to draw a circle?
  - Center – a `Point`.
  - Radius – a number (distance from center to edge).

```
eye = Circle(Point(250, 250), 200)
```

  - Center is at $(250, 250)$
  - Radius 200: top is at $(y = 50)$, bottom at $(y = 450)$.

More shapes: circles

- The `Circle` class represents a circle (unsurprisingly)
- What information is needed to draw a circle?
  - Center – a `Point`.
  - Radius – a number (distance from center to edge).

```
eye = Circle(Point(250, 250), 200)
```

  - Center is at $(250, 250)$
  - Radius 200: top is at $(y = 50)$, bottom at $(y = 450)$.
- As with `Line`, we have to **draw** the circle to display it:

```
eye.draw(win)
```

More shapes: circles

- The `Circle` class represents a circle (unsurprisingly)
- What information is needed to draw a circle?
  - Center – a `Point`.
  - Radius – a number (distance from center to edge).

```
eye = Circle(Point(250, 250), 200)
```

  - Center is at $(250, 250)$
  - Radius 200: top is at $(y = 50)$, bottom at $(y = 450)$.
- As with `Line`, we have to **draw** the circle to display it:

```
eye.draw(win)
```

Rectangles

- We could draw a rectangle already using four `Lines`.

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way. . .
  - (and we'll see another benefit shortly).

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way. . .
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way. . .
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?
  - We really only need two opposite corners
    - ★ The `graphics` libraries can figure out the other two.

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way...
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?
  - We really only need two opposite corners
    - ⋆ The `graphics` libraries can figure out the other two.

```
box = Rectangle(Point(50, 100), Point(250, 350))
```

  - What is the width?

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way...
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?
  - We really only need two opposite corners
    - ⋆ The `graphics` libraries can figure out the other two.

```
box = Rectangle(Point(50, 100), Point(250, 350))
```

  - What is the width? $(250 - 50) = 200$
  - Height?

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way...
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?
  - We really only need two opposite corners
    - ⋆ The `graphics` libraries can figure out the other two.

```
box = Rectangle(Point(50, 100), Point(250, 350))
```

  - What is the width? $(250 - 50) = 200$
  - Height? $(350 - 100) = 250$

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way...
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?
  - We really only need two opposite corners
    - ⋆ The `graphics` libraries can figure out the other two.

```
box = Rectangle(Point(50, 100), Point(250, 350))
```

  - What is the width? $(250 - 50) = 200$
  - Height? $(350 - 100) = 250$
  - We gave the upper-left and lower-right, but didn't have to:

```
box = Rectangle(Point(250, 100), Point(50, 350))
```

Rectangles

- We could draw a rectangle already using four `Lines`.
- But there's an easier way...
  - (and we'll see another benefit shortly).
- What information do we need to draw a rectangle?
  - Four corners?
  - We really only need two opposite corners
    - ⋆ The `graphics` libraries can figure out the other two.

```
box = Rectangle(Point(50, 100), Point(250, 350))
```

  - What is the width? $(250 - 50) = 200$
  - Height? $(350 - 100) = 250$
  - We gave the upper-left and lower-right, but didn't have to:

```
box = Rectangle(Point(250, 100), Point(50, 350))
```

Polygons

- We can also make a general polygon shape:
  - Any number of sides, at any angle.
  - How could we specify that?

Polygons

- We can also make a general polygon shape:
  - Any number of sides, at any angle.
  - How could we specify that?
  - List the vertices (corners)!

```
tri = Polygon(Point(100, 100), Point(300, 100),
              Point(200, 250))
```

Polygons

- We can also make a general polygon shape:
  - Any number of sides, at any angle.
  - How could we specify that?
  - List the vertices (corners)!

```
tri = Polygon(Point(100, 100), Point(300, 100),
              Point(200, 250))
```

  - `tri` would be a triangle (three points).
    - ★ You can have any number of points.

Polygons

- We can also make a general polygon shape:
  - Any number of sides, at any angle.
  - How could we specify that?
  - List the vertices (corners)!

```
tri = Polygon(Point(100, 100), Point(300, 100),
              Point(200, 250))
```

  - `tri` would be a triangle (three points).
    - ★ You can have any number of points.
  - Draws a line from the first point to the second.
  - Then the second to the third.
  - Finally, from the last point back to the first.

Polygons

- We can also make a general polygon shape:
  - Any number of sides, at any angle.
  - How could we specify that?
  - List the vertices (corners)!

```
tri = Polygon(Point(100, 100), Point(300, 100),
              Point(200, 250))
```

  - `tri` would be a triangle (three points).
    - ★ You can have any number of points.
  - Draws a line from the first point to the second.
  - Then the second to the third.
  - Finally, from the last point back to the first.
  - Order matters!
    - ★ Maybe not for a triangle, but with four or more points it does.
    - ★ Rectangle vs. bowtie.

Polygons

- We can also make a general polygon shape:
  - Any number of sides, at any angle.
  - How could we specify that?
  - List the vertices (corners)!

```
tri = Polygon(Point(100, 100), Point(300, 100),
              Point(200, 250))
```

  - `tri` would be a triangle (three points).
    - ★ You can have any number of points.
  - Draws a line from the first point to the second.
  - Then the second to the third.
  - Finally, from the last point back to the first.
  - Order matters!
    - ★ Maybe not for a triangle, but with four or more points it does.
    - ★ Rectangle vs. bowtie.

Ovals

An oval is a stretched-out circle.

- How could we specify an oval?

Ovals

An oval is a stretched-out circle.

- How could we specify an oval?
  - Several possibilities: center and two radii, two foci, ...

Ovals

An oval is a stretched-out circle.

- How could we specify an oval?
  - Several possibilities: center and two radii, two foci, ...
- The `graphics` library uses a **bounding box**.
  - Class `Oval`.

Ovals

An oval is a stretched-out circle.

- How could we specify an oval?
  - Several possibilities: center and two radii, two foci, ...
- The `graphics` library uses a **bounding box**.
  - Class `Oval`.
  - Constructor takes two `Point` arguments.
    - The corners of a rectangle (the bounding box).
    - The oval will fit in the box as tightly as possible.
    - Doesn't actually draw the bounding box!

Ovals

An oval is a stretched-out circle.

- How could we specify an oval?
  - Several possibilities: center and two radii, two foci, ...
- The `graphics` library uses a **bounding box**.
  - Class `Oval`.
  - Constructor takes two `Point` arguments.
    - ★ The corners of a rectangle (the bounding box).
    - ★ The oval will fit in the box as tightly as possible.
    - ★ Doesn't actually draw the bounding box!

```
ov = Oval(Point(100, 200), Point(400, 300))
ov.draw(win)
```

Ovals

An oval is a stretched-out circle.

- How could we specify an oval?
  - Several possibilities: center and two radii, two foci, ...
- The `graphics` library uses a **bounding box**.
  - Class `Oval`.
  - Constructor takes two `Point` arguments.
    - ★ The corners of a rectangle (the bounding box).
    - ★ The oval will fit in the box as tightly as possible.
    - ★ Doesn't actually draw the bounding box!

```
ov = Oval(Point(100, 200), Point(400, 300))
ov.draw(win)
```

Images

- The `graphics` library can draw images.
  - Supports GIF format (not JPEG!)

Images

- The `graphics` library can draw images.
  - Supports GIF format (not JPEG!)
  - Give the position and a filename:

```
pic = Image(Point(250, 250), "pic.gif")
```

Images

- The `graphics` library can draw images.
  - Supports GIF format (not JPEG!)
  - Give the position and a filename:

```
pic = Image(Point(250, 250), "pic.gif")
```

  - The image will be centered at $(250, 250)$.
  - It will be loaded from the file `pic.gif`.
    - ★ Looks in the same directory as the program.
    - ★ You can instead give an **absolute path** with a directory name.

Images

- The `graphics` library can draw images.
  - Supports GIF format (not JPEG!)
  - Give the position and a filename:

```
pic = Image(Point(250, 250), "pic.gif")
```

  - The image will be centered at $(250, 250)$.
  - It will be loaded from the file `pic.gif`.
    - ★ Looks in the same directory as the program.
    - ★ You can instead give an **absolute path** with a directory name.
    - ★ Whoever runs your program needs the image file too!

Images

- The `graphics` library can draw images.
  - Supports GIF format (not JPEG!)
  - Give the position and a filename:

```
pic = Image(Point(250, 250), "pic.gif")
```

  - The image will be centered at $(250, 250)$.
  - It will be loaded from the file `pic.gif`.
    - ★ Looks in the same directory as the program.
    - ★ You can instead give an **absolute path** with a directory name.
    - ★ Whoever runs your program needs the image file too!
- Have to use `pic.draw(win)` to display it.

Images

- The `graphics` library can draw images.
  - Supports GIF format (not JPEG!)
  - Give the position and a filename:

```
pic = Image(Point(250, 250), "pic.gif")
```

  - The image will be centered at $(250, 250)$.
  - It will be loaded from the file `pic.gif`.
    - ★ Looks in the same directory as the program.
    - ★ You can instead give an **absolute path** with a directory name.
    - ★ Whoever runs your program needs the image file too!
- Have to use `pic.draw(win)` to display it.

More methods

- So far we've seen one method for graphics shapes.

```
obj.draw(win)
```

- There are several more.

More methods

- So far we've seen one method for graphics shapes.

  ```
  obj.draw(win)
  ```

- There are several more.
- `obj.setWidth(pixels)`
  - Change the width of the shape's lines.
  - Usually do this before calling `draw`.

More methods

- So far we've seen one method for graphics shapes.

  ```
  obj.draw(win)
  ```

- There are several more.
- `obj.setWidth(pixels)`
  - Change the width of the shape's lines.
  - Usually do this before calling `draw`.
- `obj.move(dx, dy)`
  - Moves the shape by *dx* in the *x* axis, *dy* in the *y* axis.
  - *Added to* the original coordinates.

More methods

- So far we've seen one method for graphics shapes.

  ```
  obj.draw(win)
  ```

- There are several more.
- `obj.setWidth(pixels)`
  - Change the width of the shape's lines.
  - Usually do this before calling `draw`.
- `obj.move(dx, dy)`
  - Moves the shape by *dx* in the *x* axis, *dy* in the *y* axis.
  - *Added to* the original coordinates.
  - Can do this even after drawing the shape—animation!

More methods

- So far we've seen one method for graphics shapes.

```
obj.draw(win)
```

- There are several more.
- `obj.setWidth(pixels)`
  - Change the width of the shape's lines.
  - Usually do this before calling `draw`.
- `obj.move(dx, dy)`
  - Moves the shape by *dx* in the *x* axis, *dy* in the *y* axis.
  - *Added to* the original coordinates.
  - Can do this even after drawing the shape—animation!
- `obj.undraw()`
  - Erases the shape, which disappears immedately.
  - Anything "underneath" comes back!

More methods

- So far we've seen one method for graphics shapes.

```
obj.draw(win)
```

- There are several more.
- `obj.setWidth(pixels)`
  - Change the width of the shape's lines.
  - Usually do this before calling `draw`.
- `obj.move(dx, dy)`
  - Moves the shape by *dx* in the *x* axis, *dy* in the *y* axis.
  - *Added to* the original coordinates.
  - Can do this even after drawing the shape—animation!
- `obj.undraw()`
  - Erases the shape, which disappears immedately.
  - Anything "underneath" comes back!

Color methods

- Shapes have two different colors: the **fill** and **outline**.

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.
  - The **outline color** is used for the border.
    `line.setOutline('red')`

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.
  - The **outline color** is used for the border.
    `line.setOutline('red')`
    - ★ For a Line or Point, that's the whole thing.

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.
  - The **outline color** is used for the border.
    `line.setOutline('red')`
    - ★ For a Line or Point, that's the whole thing.
  - The window as a whole has a **background color**.
    `win.setBackground('yellow')`

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.
  - The **outline color** is used for the border.
    `line.setOutline('red')`
    - ★ For a Line or Point, that's the whole thing.
  - The window as a whole has a **background color**.
    `win.setBackground('yellow')`
- The color names can be a bit obscure ("firebrick"? "purple4"?)
  - `http://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm`
  - `http://wiki.tcl.tk/37701`

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.
  - The **outline color** is used for the border.
    `line.setOutline('red')`
    - ★ For a Line or Point, that's the whole thing.
  - The window as a whole has a **background color**.
    `win.setBackground('yellow')`
- The color names can be a bit obscure ("firebrick"? "purple4"?)
  - `http://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm`
  - `http://wiki.tcl.tk/37701`
  - Or specify red-green-blue values:

```
line.setOutline(color_rgb(255, 128, 0))  # orange
```

Color methods

- Shapes have two different colors: the **fill** and **outline**.
  - The **fill color** is used for the "inside" of the shape.
    `box.setFill('blue')`
    - ★ Specify the color name as a string.
    - ★ Points and lines don't have an inside.
    - ★ This is why Rectangle and Polygon are more than just Lines.
  - The **outline color** is used for the border.
    `line.setOutline('red')`
    - ★ For a Line or Point, that's the whole thing.
  - The window as a whole has a **background color**.
    `win.setBackground('yellow')`
- The color names can be a bit obscure ("firebrick"? "purple4"?)
  - `http://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm`
  - `http://wiki.tcl.tk/37701`
  - Or specify red-green-blue values:

```
line.setOutline(color_rgb(255, 128, 0))  # orange
```

Recap: object-oriented programming terminology

- **Object**: A thing that can be stored in a variable.

Recap: object-oriented programming terminology

- **Object**: A thing that can be stored in a variable.
- **Class**: A *type* that represents a particular kind of thing.
  - A template for making **objects** of that type.
  - `GraphWin`, `Line`, `str`, ... are classes.
  - The object `"Hello"` **belongs to** the class `str`.

Recap: object-oriented programming terminology

- **Object**: A thing that can be stored in a variable.
- **Class**: A *type* that represents a particular kind of thing.
  - A template for making **objects** of that type.
  - `GraphWin`, `Line`, `str`, ... are classes.
  - The object `"Hello"` **belongs to** the class `str`.
- **Constructor**: A function that creates an object belonging to a class.
  - Has the same name as the class.
  - Uses the class template to "stamp out" a new object.
  - `Point(100, 100)` is a constructor call.

Recap: object-oriented programming terminology

- **Object**: A thing that can be stored in a variable.
- **Class**: A *type* that represents a particular kind of thing.
  - A template for making **objects** of that type.
  - `GraphWin`, `Line`, `str`, ... are classes.
  - The object `"Hello"` **belongs to** the class `str`.
- **Constructor**: A function that creates an object belonging to a class.
  - Has the same name as the class.
  - Uses the class template to "stamp out" a new object.
  - `Point(100, 100)` is a constructor call.
- **Method**: A function that belongs to an object and does something to or with the object.
  - In `myline.draw(win)`, `draw` is a method of the `Line` class.
    - (*not* of the `GraphWin` class!)
  - Methods are defined by classes and work on any object of that class.

Recap: object-oriented programming terminology

- **Object**: A thing that can be stored in a variable.
- **Class**: A *type* that represents a particular kind of thing.
  - A template for making **objects** of that type.
  - `GraphWin`, `Line`, `str`, ... are classes.
  - The object `"Hello"` **belongs to** the class `str`.
- **Constructor**: A function that creates an object belonging to a class.
  - Has the same name as the class.
  - Uses the class template to "stamp out" a new object.
  - `Point(100, 100)` is a constructor call.
- **Method**: A function that belongs to an object and does something to or with the object.
  - In `myline.draw(win)`, `draw` is a method of the `Line` class.
    - (*not* of the `GraphWin` class!)
  - Methods are defined by classes and work on any object of that class.

Next time

- More graphics.
- Making decisions: if statements.