As you arrive:

1. Start up your computer and plug it in
2. ***Log into Angel*** and go to CSSE 120
3. Do the ***Attendance Widget*** – the PIN is on the board
4. Go to the course ***Schedule Page***
5. Open the ***Slides*** for today if you wish
6. Check out today's project: `08-DebuggingObjectsAndGraphics`

Plus in-class time working on these concepts, continued as homework.

Debugging

- What debugging includes
- Two ways to debug:
  - Using *print* statements
  - Using a *debugger*
- Debugging tips

Objects & Graphics

- The object of objects
- Interaction among objects
- Graphical objects
- Mouse events

Checkout today's project:
`08-DebuggingObjectsAndGraphics`

Troubles getting today's project? If so: →

Are you in the Pydev perspective? If not:

- **`Window ~ Open Perspective ~ Other`** then **`Pydev`**

Messed up views? If so:

- **`Window ~ Reset Perspective`**

No SVN repositories view (tab)? If it is not there:

- **`Window ~ Show View ~ Other`** then **`SVN ~ SVN Repositories`**

In your SVN repositories view (tab), expand your repository (*the top-level item) if not already expanded.*

- If no repository, perhaps you are in the wrong Workspace. Get help as needed.

Right-click on today's project, *then select* ***`Checkout`****. Press* ***OK*** *as needed.*

The project shows up in the **`Pydev Package Explorer`** to the right. Expand and browse the modules under **`src`** as desired.

Outline

Check out today's project:
`08-DebuggingObjectsAndGraphics`

- **Questions?**
- **Debugging**
  - What debugging includes
  - Two ways to debug:
    - *print* statements
    - *debugger*
  - Using the Debugger in Eclipse
  - Debugging tips
- **Objects**
  - What are objects? Why are they useful?
  - How do you *construct* an object?
  - What do objects have? *Fields*
  - What can objects do? *Methods*
  - Interaction among objects. *UML class diagrams*
- **Examples of objects**
  from *zellegraphics*
  - GraphWin, Point, Line, Circle. Mouse events.

Plus in-class time working on these concepts, continued as homework.

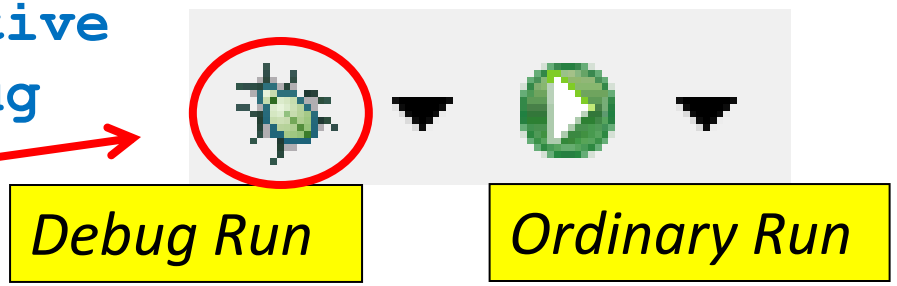
Debugging

- Debugging includes:
  - Discovering errors
  - Coming up with a hypothesis about the cause
  - Testing your hypothesis
  - Fixing the error
- Ways to debug
  - Insert print statements to show program flow and data
  - Use a debugger:
    - A program that executes another program and displays its runtime behavior, step by step
    - Part of every modern IDE

Q1

Using a Debugger

- Typical debugger commands:
  - Set a breakpoint—place where you want the debugger to pause the program
  - Single step—execute one line at a time
  - Inspect a variable—look at its changing value over time
- Debugging example. In today's project in Eclipse:
  - Briefly examine the `01-MoveCircle.py` module.
  - In that module, start a *debugging session* in the *Debug perspective*:
    - **`Window → Open Perspective → Other, then Debug`**
    - Click **Debug Run** icon (top-left side of work bench)



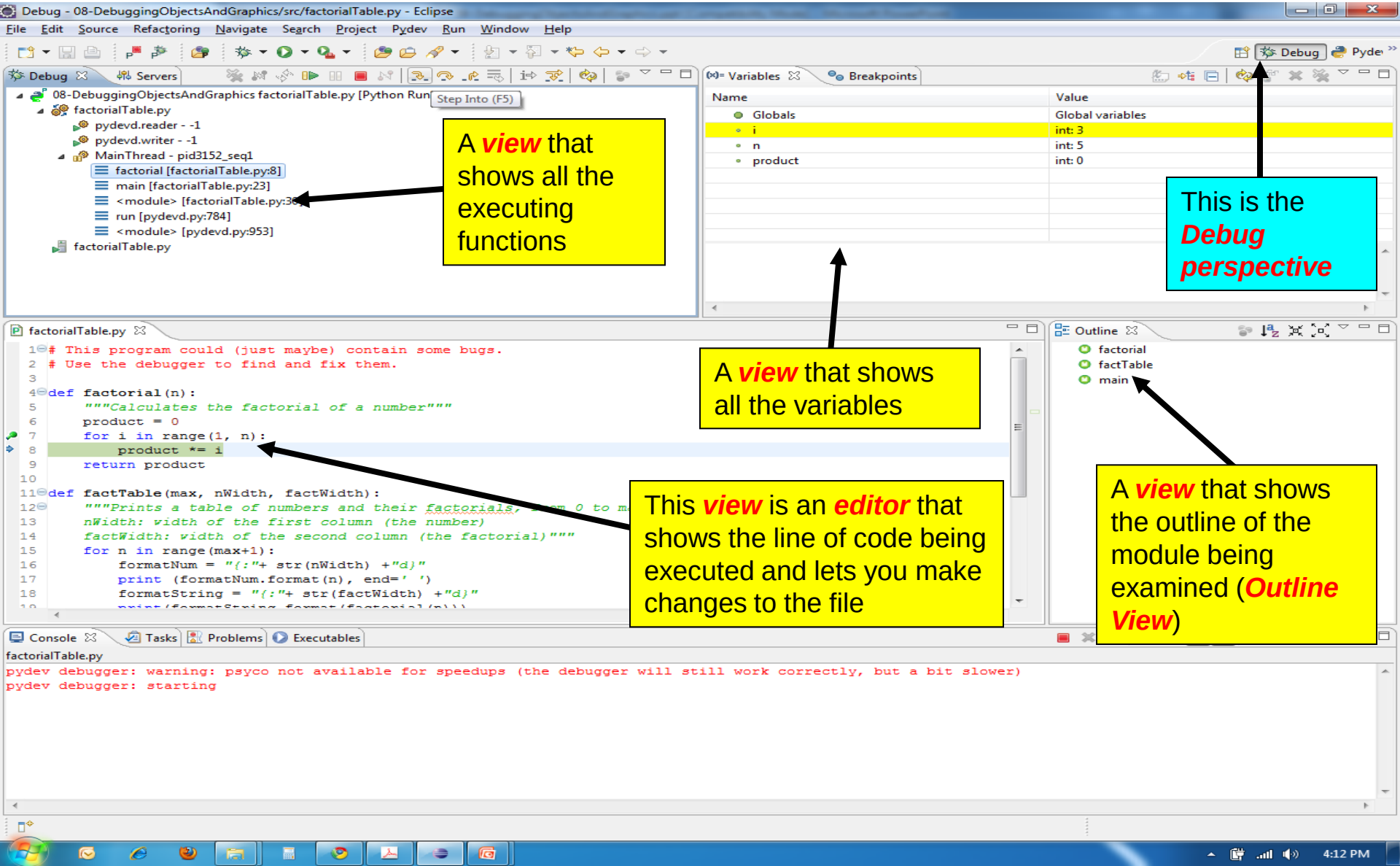
Learn how to, in the Debugger:

1. Start a *debugging session* in the *Debug Perspective*.
   - Switch back and forth between the *Debug* and *Pydev* perspectives.
2. Set *breakpoints* in your code.
   - And unset them.
3. *Inspect* the variables in the current scope at a breakpoint.
   - See their current *values* and *types*.
   - See *which ones have changed* since the last breakpoint.
   - Expand them to see their *fields* and the fields' values.
4. *Debug Run* in the Debug Perspective:
   - *Resume*, continuing to the next breakpoint
   - *Single-Step* to the next statement
   - At a function call, *Step-Over* it
   - Inside a function, *Step-Return* from it

Using the debugger in Eclipse

- Set a breakpoint
  - Double click in left margin of **editor view**
- Step over (when you know a function works)
  - Click step-over icon or use F6 key
- Variable inspection
  - Look at the new value of i, cir after each time though the loop

Sample Debugging Session: Eclipse



Tips to Debug Effectively

- Reproduce the error
- Simplify the error
- Divide and conquer
  - Set a breakpoint and inspect: does the error occur before the breakpoint or after?
- Know what your program should do
- Look at the details
  - Compare the actual content of variables against the values that you think they should have.
  - This often "wakes you up" into reading what is actually written in the code instead of what you intended to write.
- ***Understand*** each bug before you fix it
- Practice!

Use the scientific method:

- hypothesize
- experiment
- fix bug
- repeat experiment

Outline of next part of this session

- **Objects**
  - What are objects? Why are they useful?
  - How do you *use objects*?
    - How do you *construct* an object?
    - What do objects have? *Fields*
    - What can objects do? *Methods*
  - Interaction among objects. *UML class diagrams.*
- **Examples of objects** from *zellegraphics*
  - GraphWin, Point, Line, Circle
  - Mouse events

What are objects?

- ***Traditional*** view, in languages like C
  - Data types are *passive*
    - They have values
    - There are operations that act on the data types
      - The data type itself cannot do anything
- ***Object-oriented*** view, in languages like Python (and most other modern languages)
  - Have *objects*, which are *active* data types. Objects:
    - ***Know stuff – they contain data***
      - The data that an object holds are its ***instance variables*** (aka ***fields***)
    - ***Can do stuff – they can initiate operations***
      - The operations that an object can do are its ***methods***

Q2

Traditional, non-object-oriented, design

- Break the problem into subproblems. That is:
  - To solve the problem I need to do: A, B, C, …
    - To solve A, I need to do: A1, A2, A3, …
      - To solve A1, I need to do A1a, A1b, A1c, …
      - To solve A2, I need to do A2a, A2b, A2c, …
        - etc
    - To solve B, I need to do: B1, B2, B3, …
    - etc, until the units are so small that you can just *do* them
- The units become *functions*
- This process is called *procedural decomposition*

Q3

Modern, *object-oriented*, design

- Basic idea of object-oriented (OO) development
  - *View a complex system as interaction of simple objects*
- In doing OO development, ask:
  1. What ***things*** (**objects**) are involved in the solution to my problem?

     The *types* of those things become our ***classes***

     These *things* often come from *nouns* in the problem description, e.g.
     single concepts *visual elements*
     abstractions of real-life entities
     actors utilities
  2. For each type of thing (i.e., each *class*), what ***responsibilities*** does it have?

     What can it do? E.g. *A list can append stuff to itself.*

     These responsibilities become the *methods* of that class: `append`

     These *responsibilities* often come from *verbs* in the problem description
  3. To carry out those responsibilities:
     a. What other objects does it need help from? *Relationships between classes*
     b. What objects does it have within? Become the *instance variables* of the class.

Q4-6

Why is the *object-oriented* view useful?

- Procedural decomposition is useful and forms an important part of OO design
- But for complex systems, we often find it easier to think about the complex system as the interaction of simple objects than to just "break it down into its parts"
- In practice, most complex software systems today are designed using OO design

How do you *use* objects?

Recall that objects:
- Know stuff (*fields*)
- Can do stuff (*methods*)

- To *construct* an object:

```
win = GraphWin()
p1 = Point(500, 450)
line = Line(p1, Point(30, 40))
circle = Circle(p1, 100)
```

Constructor:
- Call it like a function, using the name of the ***class***
- Uniform style: Class names begin with an uppercase letter
- The constructor allocates space for the object and does whatever initialization the class specifies

- To *ask an object to do something*, i.e. to apply its *methods* to it:

```
p1.draw(win)
line.move(45, -60)
x = p1.getX()
center = circle.getCenter()
```

Method call:
- Use the ***dot notation:***

`Who.doesWhat(withWhat)`

Just like a function call, except that the method has access to the object invoking the method.
- So the object is an *implicit argument* to the method call

- To reference what the object knows (its *instance variables*): `p1.x` `circle.p1` `circle.p2`

Instance variable reference:
- Use the ***dot notation*** but without parentheses

`Who.hasWhat`

How do objects interact?

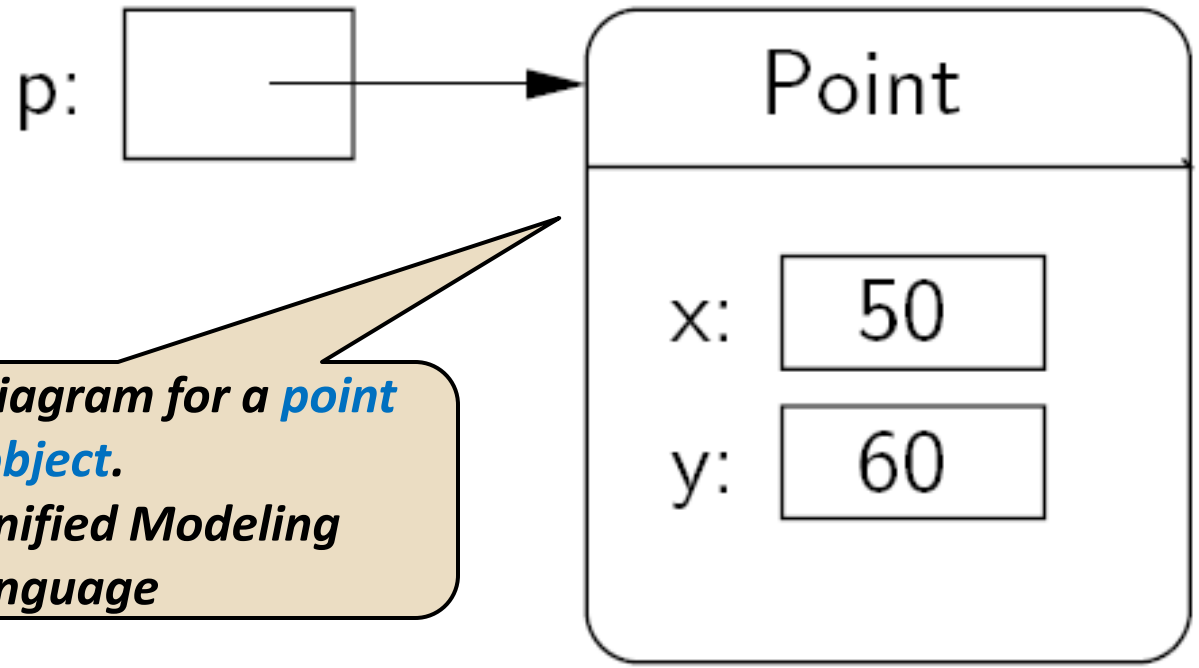
- Objects interact by sending each other messages
  - Message: request for object to perform one of its operations
  - Example: the brain can ask the feet to walk
  - In Python, messages happen *via* method calls.

```
win = GraphWin()      # constructor
p = Point(50, 60)     # constructor
p.getX()              # accessor method
p.getY()              # accessor method
p.draw(win)           # method
```

Q7-8

How do objects interact? Point

p = Point(50, 60)

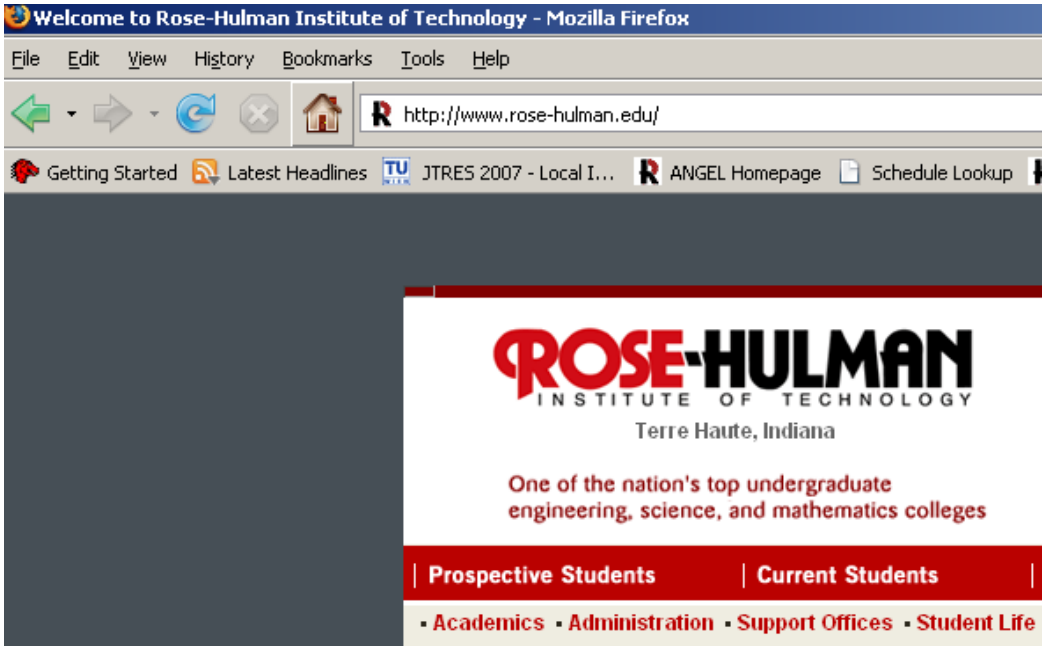


UML object diagram for a point object.
UML ➔ Unified Modeling Language

Q9-10

Simple graphics programming

- Graphics is fun and provides a great vehicle for learning about objects
- Computer Graphics: study of graphics programming
- Graphical User Interface (GUI)



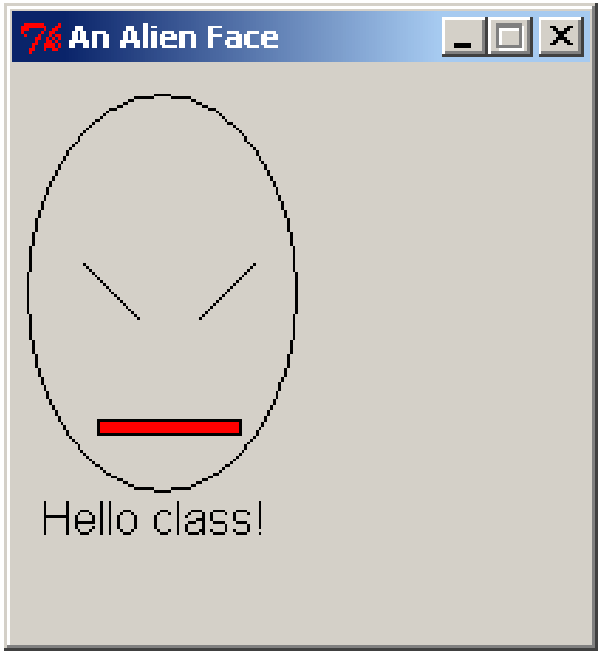
Q11

Review: You choose how to import

- Must import graphics library before accessing it
  - >>> import zellegraphics
  - >>> win = zellegraphics.GraphWin()
- Another way to import graphics library
  - >>> from zellegraphics import *
  - win = GraphWin()

Using graphical objects

- Using different types of objects from the graphics library, draw the following alien face and message in the 03-alienFace.py module



Q12

Paige clearly isn't working on homework for CSSE120

- Preview of tonight's homework:
  1. Read in and draw cool plots from the points in the files you generated in HW6 and 7
  2. Create a cool slideshow picture viewer!

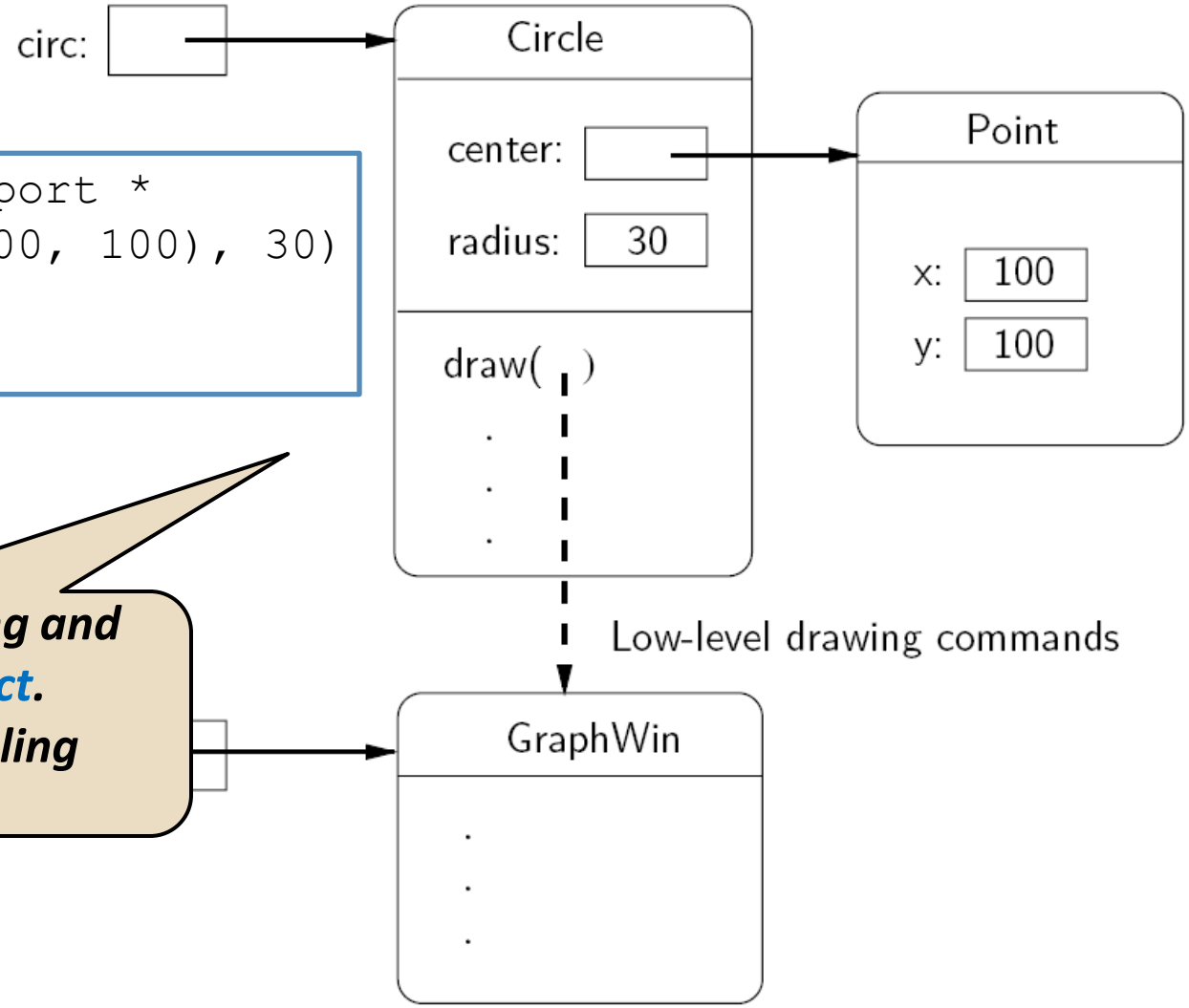
Review: Class and object terminology

- Different types of objects
  - Point, Line, Rectangle, Oval, Text
  - These are examples of *classes*
- Different objects
  - head, leftEye, rightEye, mouth, message
  - Each is an *instance* of a class
  - Created using a *constructor*
  - Objects have *instance variables* (called *fields* in some languages)
  - Objects use *methods* to operate on instance variables
    - *Accessor methods* return data from the object

Q13 - 14

Object interaction to draw a circle

```
from zellegraphics import *
circ = Circle(Point(100, 100), 30)
win = GraphWin()
circ.draw(win)
```



UML diagram for creating and drawing a circle object.
UML ➔ Unified Modeling Language

Interactive graphics

- *GUI*—Graphical User Interface
  - Accepts input
    - Keyboard, mouse clicks, menu, text box
  - Displays output
    - In graphical format
    - On-the-fly
- Developed using *Event-Driven Programming*
  - Program draws interface elements (*widgets*) and **waits**
  - Program responds when user does something

Q15

getMouse

- `win.getMouse()`
  - Causes the program to **pause, waiting** for the user to click with the mouse somewhere in the window
  - To find out where it was clicked, assign it to a variable:
    - `p = win.getMouse()`

Q16-17

Mouse Event Exercise

Together, lets' solve the following problem:

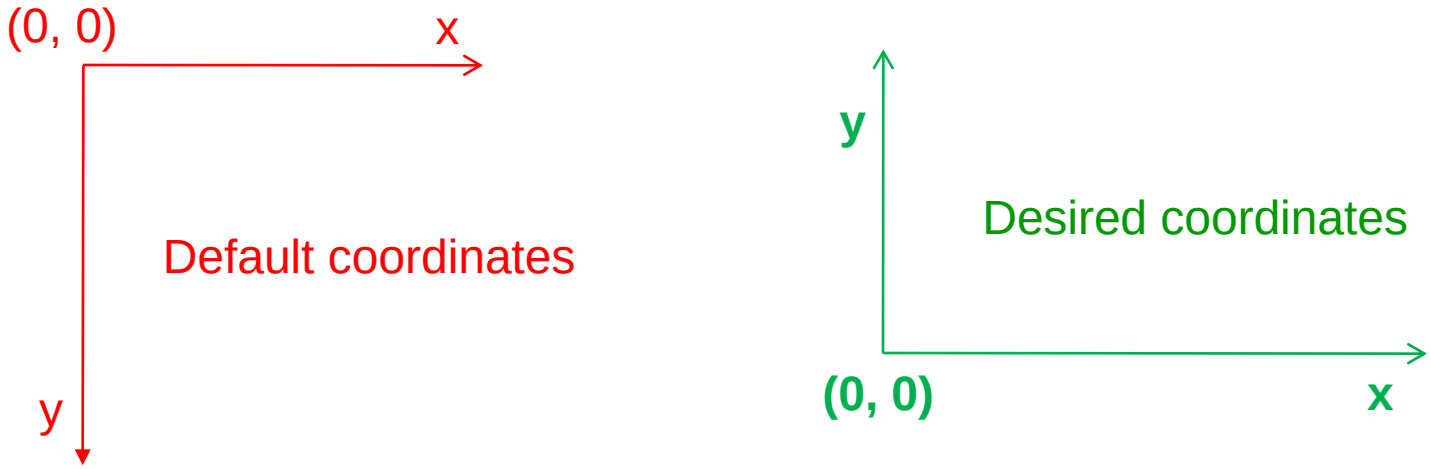
Create a program in module, `04-clickMe.py`, with a window labeled "Click Me!" that displays the message *You clicked (x, y)* to the console the first 5 times the user clicks in the window.

The program also draws a red-filled circle, with blue outline, in the location of each of these first 5 clicks.

The program closes the window on the 6th click

Coordinate systems

- An important use of graphics is to represent **data** visually
  - Example: a bar chart
- We really want (0,0) to be in the lower-left corner



Desired coordinate system

- `win.setCoords(x1, y1, x2, y2)` **method from** `GraphWin` **class**
  - Sets the coordinates of the window to run from (x1,y1) in the lower-left corner to (x2,y2) in the upper-right corner.