As you arrive:

1. Start up your computer and plug it in.
2. ***Log into Angel*** and go to CSSE 120.
   Do the ***Attendance Widget*** –
   the PIN is on the board.
3. Go to the ***Course Schedule*** web page.
   Open the ***Slides*** for today if you wish.
4. Checkout today's project: `Session04_NumbersAndLoops`

Session 4

Loops (Counted and Accumulator)

Review

- Input-compute-output
- Functions: defining and calling
  - With parameters
  - Called with actual arguments
  - Returning values

Using a Debugger

Loops

- Counted loops
  - FOR loops with RANGE expressions
- Accumulator loops

Checkout today's project:

`Session04_NumbersAndLoops`

Are you in the ***Pydev*** ***perspective? If not:***

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

Messed up views? If so:

`Window ~ Reset Perspective`

Troubles getting today's project? If so:

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

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

1. ***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.
2. ***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 of today's session

Checkout today's project:
`Session04_NumbersAndLoops`

- ***Review***
  - Organizing a program into *functions.* How to:
    - *Define* a function. *Call* a function. Start a program in *main*
  - The *input-compute-output* pattern
  - Functions with *parameters* that *return* values. ***An exercise for more practice.***
- ***Using a Debugger***
  - Why, How
- ***Loops***
  - ***Counted*** loops
  - ***Accumulator*** loops

Practice, practice, practice!

- **Functions**
  - Writing with parameters
  - Calling with arguments
  - Returning values, using them
- Using **objects**: in zellegraphics
  - The dot notation, revisited
- **Loops**

The code in these examples appears in full in the `m1_defining_and_calling_functions.py` module.

Review: Organizing a program into *functions*

- ***Define*** a function:

```
def hello():
    """ Prints a greeting. """
    print('Hello, World!')
```

Just DEFINES what the function does. Doesn't "do" anything of itself. **Note:**

- `def` ***keyword***
- ***Parentheses***
- ***Colon***
- ***Indented body***
- ***Documentation-comment***

- ***Call*** (aka ***invoke***) a function:

```
def main():
    """ Prints a greeting. """
    hello()
```

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

These are function CALLS:

- ***To the built-in*** *print* ***function***
  Note:
  - *Use of* ***actual argument*** *here*
  - *All calls require parentheses, even if nothing is in them*
- ***To the above-defined*** *hello* ***function***
- ***To the above-defined*** *main* ***function***

So ***main*** *runs when the module runs*

Questions?

Q1-2

The code in these examples appears in full in the `m2_input_compute_output.py` module.

Review: The *input-compute-output* pattern

```
def celsius_to_fahrenheit():
    celsius = float(input('What is Cel. temperature? '))
    fahrenheit = 9/5 * celsius + 32
    print('Temperature is', fahrenheit, 'degrees Fahr.')
```

- Getting input from the user
  - `input('What is Cel. temperature? ')`
  - `float(...)` and `int(...)`
  - `celsius = ...`
- Computing a value using an assignment
  - `fahrenheit = 9/5 * celsius + 32`
- Printing values to the console
  - `print('Tem...', fahrenheit, 'deg...')`

Questions?

Q3-5

The code in these examples appears in full in the **m3_parameters_arguments_and_return.py** module.

Review: formal ***parameters*** & actual ***arguments***

The returned value is ***captured*** *in variable* f

Note: This example omits documentation-comments and uses uninformative variable names (c and f) in order to make things fit on the slide. See the module in today's project for this same example done more completely.

```
def main():
    for c in range(0, 101, 10):
        f = celsius_to_fahrenheit(c)
        print(c, 'degrees Celsius is',
              f, 'degrees Fahrenheit')


def celsius_to_fahrenheit(celsius):
    fahrenheit = (9 / 5) * celsius + 32
    return fahrenheit
```

The actual argument c

The formal parameter celsius

The computed value is ***RETURNED*** *(not printed) here*

A local variable fahrenheit

Do you see how the parameter makes the function *powerful*?
Questions?

The names ***celsius*** and ***fahrenheit*** are ***local*** to their function.
They have NOTHING to do with any uses of those names in ***main*** or elsewhere.

Q6-7

Exercise: Parameters, revisited

- Here is an outline of what you will do in this exercise:
  - ***Step 1: Briefly revisit objects***, including how to:
    - ***Construct*** an object
    - Apply a ***method*** to an object, using the ***dot notation***
    - Reference an ***instance variable*** (aka ***field***) of an object, using the ***dot notation***
  - ***Step 2: Introduce using a debugger***
    - Why it is helpful
    - How to use our debugger to:
      - Set ***breakpoints*** in your code and then start a debugging session.
      - In the debugging session, ***step through*** lines of code and ***inspect*** variables.
  - ***Step 3: Practice functions with parameters***
    - Implement three ***distance*** functions.
    - Call those functions with ***actual arguments***.

Step 1: Briefly revisit objects

See the next slide for more examples

- With your instructor:
  - Open `m4_distance_between_clicks.py` and run it
  - Discuss the overall structure of the program briefly
  - Discuss ***show_distances*** briefly, to revisit how to:
    - ***Construct*** an object

```
window = zg.GraphWin('Mouse-click distances', 300, 500)
```

    - Apply a ***method*** to an object, using the ***dot notation***

```
point1 = window.getMouse()
```

    - Reference an ***instance variable*** (aka ***field***) of an object, using the ***dot notation***

```
point1.x      point1.y
```

Q8

Step 1: Briefly revisit objects

`window` *references the GraphWin object*

Constructs a `zg.GraphWin` *object. Capital-G says constructor.*

```
window = zg.GraphWin('Mouse-click distances', 300, 500)
```

The code for this function shows that it returns a `zg.Text` *object*

```
text_box1 = make_text_box_centered_at(50, window)

while True:
    point1 = window.getMouse()
```

Applies the `getMouse` *method to* `window`*. Uses* `point1` *to reference the* `zg.Point` *object that getMouse returns.*

Who-dot-what-with-what notation

```
    text_box1.setText(point1)
```

Applies the `setText` *method to the* `text_box1`

```
    point2 = window.getMouse()

    point_as_string = '(' + str(point2.x) + ', '
                      + str(point2.y) + ')'
```

References the `x` *and* `y` *instance variables (aka fields) of* `point2`*.*

Q9

Step 2: Introduce using a debugger

- ***Debugging*** includes:
  - Discovering errors
  - Developing a hypothesis about the cause(s)
  - Testing your hypothesis (and revising it as needed)
  - Fixing the error
    - Using your hypothesis to determine the fix
    - Testing the fix to be sure it really fixes the error(s)
- ***Ways to debug***
  - Insert `print` statements to show program flow and data
  - Use a ***debugger***:
    - A program that executes another program and displays its run-time behavior, step by step
    - Part of every modern IDE (including Eclipse)

Learn how to, in the Debugger:

1. Set (and unset) ***breakpoints***
2. Start a ***debugging session*** in the ***Debug Perspective***
   - ***Debug Run*** to the next breakpoint
   - Switch back and forth between the ***Debug*** and ***Pydev*** perspectives
3. ***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
4. ***Inspect*** the variables in the current scope at a breakpoint
   - See their current ***values*** and ***types***
   - See ***which have changed*** since the last breakpoint
   - Expand them to see their ***instance variables*** (aka ***fields***) and values

Your instructor will show you how to do this, live in Eclipse, in `m4_distance_between_clicks.py`
The next slides summarize what your instructor will show you.

To start/end a ***debugging session***

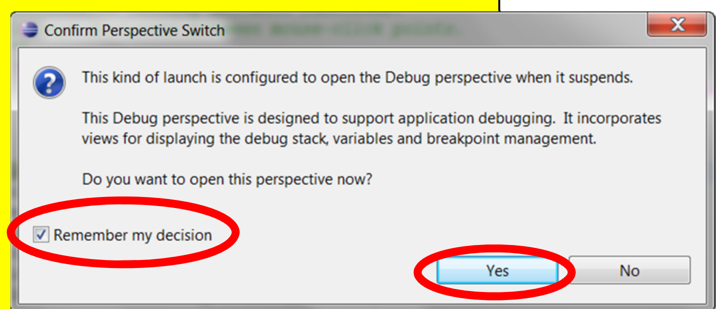
- To start a ***debugging session*** in the **Debug Perspective**:

Debug Run

Ordinary Run

Click the ***Debug*** button on the ToolBar
and (if asked) select ***Debug As*** … ***Python Run***

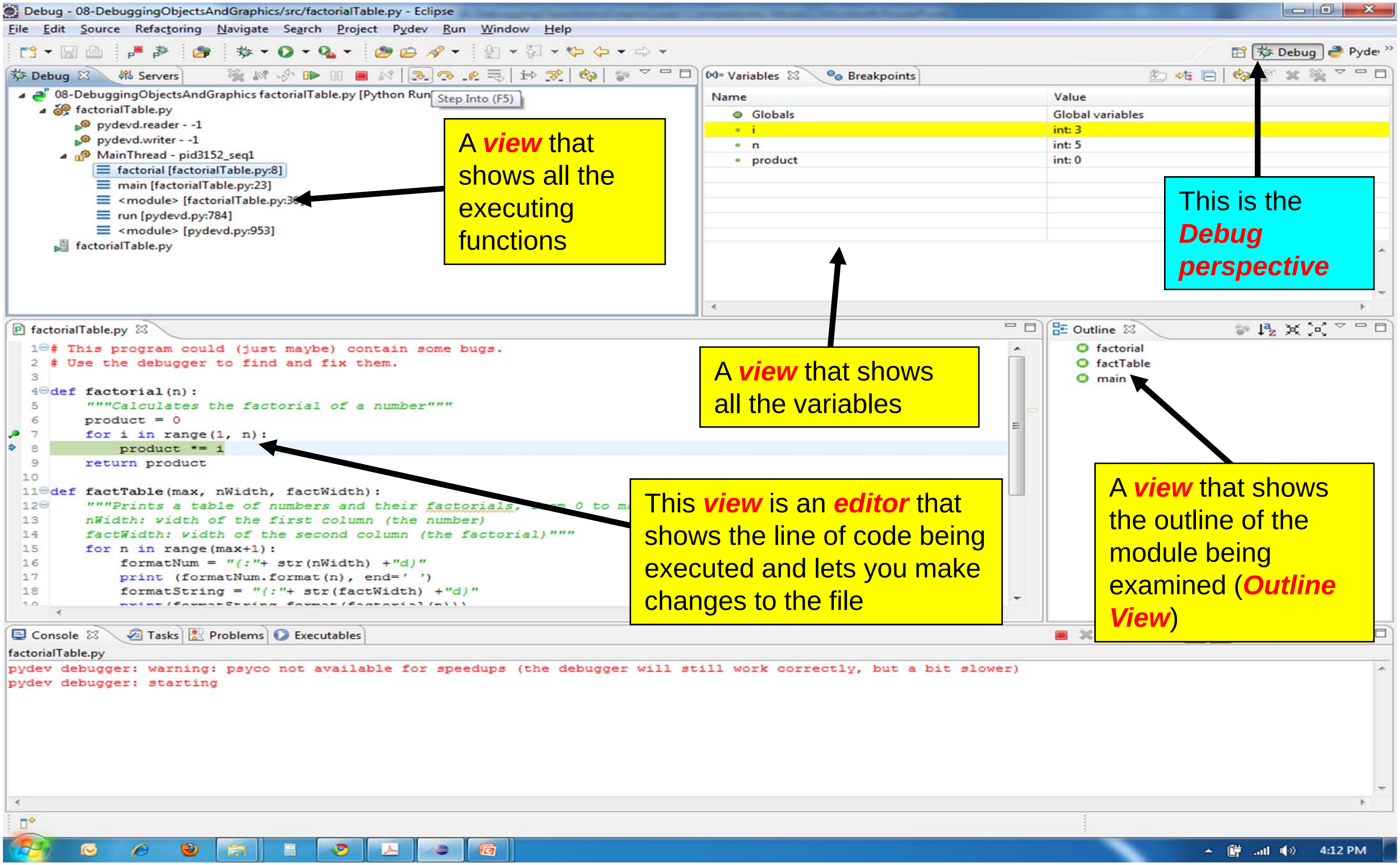
If asked to **Confirm Perspective Switch** to open the Debug perspective
- Check the **Remember my decision** box
- Press ***Yes***



- To switch between ***Debug*** and ***Pydev*** perspectives:

Click the ***Pydev*** and/or ***Debug*** buttons in the ***upper-right*** corner of Eclipse, or select the ***Open Perspective*** button there.

Sample Debugging Session: Eclipse



Step 3: *Practice functions with parameters*

- Do the TODO's in the module
- They will ask you to:
  - Implement three ***distance*** functions
  - Call those functions with ***actual arguments***

Exercise: Counted Loops

- Open `m5_counted_loops.py`
- With your instructor, run and study the existing code

A counted loop. The `range` statement makes `k` take on values 0, 1, 2, …. 9

```
for k in range(10):
    a = 0
    b = 0
    print("{:1} {:3} {:3}".format(k, a, b))
```

Does formatted printing. The three items printed (`k,a,b`) are printed in fields of widths 1, 3 and 3, respectively.
We'll learn more about formatted printing later.

- Do the TODO's,
  using the quiz questions to guide your work.
  - Your instructor will get you started on this.

Q10-11

Exercise: Accumulator Loops

- Open **m6_accumulator_loops.py**
- With your instructor, run and study the existing code, then do the TODO's.

```
def accumulate_a_sum(n):
    """ Returns the sum  1 + 2 + 3 + ... + n  for given n. """
    sum = 0
    for k in range(1, n + 1):
        sum = sum + k

    return sum
```

The accumulator pattern:

1. ***Before the loop, initialize the accumulator variable:*** `blah = ...`
2. ***Inside a loop, accumulate with a statement like:***
   `blah = blah ...`
3. ***After the loop, the accumulator variable contains the accumulated value.***

Rest of Session

- ***Check your Quiz answers*** versus the solution
  - An assistant may check your Quiz to ensure you are using the Quizzes appropriately
- ***Work on today's homework***
  - Ask questions as needed!
- **Sources of help after class:**

  ```
  CSSE lab: Moench F-217
            7 to 9 p.m.
  Sundays thru Thursdays
  ```

  - **Assistants in the CSSE lab**
    - And other times as well (see link on the course home page)
  - **Email** `csse120-staff@rose-hulman.edu`
    - You get faster response from the above than from just your instructor