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: `Session13_LoopPatterns`

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

Loop Patterns

- For loop pattern
- While loop patterns
- Loop-and-a-half loop pattern
- File loop pattern

Practice:

with loop patterns

Checkout today's project: `Session13_LoopPatterns`

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.

Recap: Two main types of loops

- Definite Loop
  - The program knows ***before the loop starts*** how many times the loop body will execute
  - Implemented in Python as a `for` loop. Typical patterns include:
    - Counting loop, perhaps in the Accumulation Loop pattern
    - Loop through a sequence directly
    - Loop through a sequence using indices
  - Cannot be an infinite loop
- Indefinite loop
  - The body executes as long as some condition is `True`
  - Implemented in Python as a `while` statement
  - Can be an infinite loop if the condition never becomes `False`
  - Python's `for line in file:` construct
    Indefinite loop that looks syntactically like a definite loop!

Recap: Definite Loops

- ***Definite loop***
  The program knows ***before the loop starts*** how many times the loop body will execute
  - ***Counted loop***
    Special case of definite loop where the sequence can be generated by `range()`
- Implemented in Python as a `for` loop
- Example to the right shows 3 typical patterns

Examples of ***definite loops*** :

- All three of these examples illustrate the Accumulation Loop pattern
- The first example is a ***counted*** loop
- The second and third examples are equivalent ways to loop through a sequence
  - Second example is NOT a counted loop
  - Third example IS a counted loop

```
sum = 0
for k in range(10):
    sum = sum + (k ** 3)

sum = 0
for number in listOfNumbers:
    sum = sum + number

sum = 0
for k in range(len(listOfNumbers)):
    sum = sum + listOfNumbers[k]
```

Recap: Indefinite Loops

- Number of iterations is ***not known*** when loop starts
- Is typically a conditional loop
  - Keeps iterating as long as a certain condition *remains* **`True`**
  - The conditions are ***Boolean expressions***
- Typically implemented using a **`while`** statement

```
sum = 0
for k in range(10):
    sum = sum + (k ** 3)
```

Definite loop

```
sum = 0
k = 0
while k < 10:
    sum = sum + (k ** 3)
    k = k + 1
```

Indefinite loop that computes the same sum as the definite loop

Outline of Loop Patterns

- The ***compute-in-a-loop*** pattern
- Six basic *compute-in-a-loop* patterns:
  - For loop
  - While loop
    - Interactive loop
    - Sentinel loop using a special value as the sentinel
    - Sentinel loop using no-input as the sentinel
  - Loop-and-a-half
    - Combined with use of no-input as the sentinel
  - File loop
  - Nested loops (next session)
  - Wait-for-event loop (next session)

Q1

Loop patterns

- We have seen the *input-compute-output* pattern:

```
get data
compute using the data
print the result
```

Input from the user or as a parameter

Or *return* the result

- A cousin that pattern is the *compute-in-a-loop* pattern:

```
pre-loop computation
repeatedly:
        get data
        compute using the data
post-loop computation
```

We've seen a special case of this pattern: the ***Accumulator Loop*** pattern. Today we will examine other special cases.

Six basic compute-in-a-loop patterns

For loop

```
pre-loop computation
for [amount of data] :
        get data
        compute using the data
post-loop computation
```

While loop

```
pre-loop computation
while [there is more data]:
        get data
        compute using the data
post-loop computation
```

Loop and a Half

```
pre-loop computation
while True:
        get data
        if data signals end-of-data:
                break
        compute using the data
post-loop computation
```

File loop

```
pre-loop computation
for line in file:
        get data from line
        compute using the data
post-loop computation
```

Next time → **Nested loops**

Wait-for-event loop

For loop pattern →

pre-loop computation
for [amount of data] :
    get data
    compute using the data
post-loop computation

- Open the

  `module1_averageUserCount.py`

  module and execute it together
- When does the loop terminate?
- Is this the best way to make the user enter input?
  - Why?
  - Why not?

This approach is a lousy way to get numbers that the user supplies, because:

The user has to count in advance how many numbers they will supply.

While loop pattern #1

pre-loop computation
while [there is more data]:
    get data
    compute using the data
post-loop computation

- One version: an ***interactive*** loop

set a flag indicating that there is data
other pre-loop computation
while [there is more data]:
    get data
    compute using the data
    ask the user if there is more data
post-loop computation

Examine and run the *module2_averageMoreData.py* module in the project you checked out today.

This approach is also a lousy way to get numbers that the user supplies, because:

The user has to answer repeatedly the "more numbers?" question.

Q2

While loop pattern #2

pre-loop computation
while [there is more data]:
    get data
    compute using the data
post-loop computation

- Better version:
  use a ***sentinel***

get data
other pre-loop computation
while [data does not signal end-of-data]:
    compute using the data
    get data
post-loop computation

Examine and run the *module3_averageSentinel.py* module in the project you checked out today.

User signals end of data by a special "sentinel" value.

Note that the sentinel value is not used in calculations.

This approach (using negative numbers as the sentinel) has a flaw. What is that flaw?

Answer: You cannot have negative numbers included in the average!

Q3

While loop pattern #3

```
pre-loop computation
while [there is more data]:
        get data
        compute using the data
post-loop computation
```

- Best (?) version:
  use *no-input* as the ***sentinel***

```
get data as a string
other pre-loop computation
while [data is not the empty string]:
        data = float(data)
        compute using the data
        get data as a string
post-loop computation
```

Examine and run the ***module4_averageOtherSentinel.py*** module in the project you checked out today.

User **signals** end of data by pressing the **Enter key** in response to a *input*.

The ***sentinel*** value is again not used in calculations.

Above converts the data to a *float*, but other problems might do other conversions.

Loop-and-a-half pattern

- Use a ***break***

```
pre-loop computation
while True:
        get data
        if data signals end-of-data:
                break
        compute using the data
post-loop computation
```

```
pre-loop computation
while True:
        get data as a string
        if data == "":
                break
        data = eval(data)
        compute using the data
post-loop computation
```

The ***break*** command exits the enclosing loop.

Examine and run the ***module5_averageLoopAndAHalf.py*** module in the project you checked out today.

Here we continue to use no-input as the sentinel.

This pattern is equivalent to the pattern on the preceding slide. Some prefer one style; others prefer the other. You may use whichever you choose.

Q4

Escaping from a loop

- **break** statement ends the loop immediately
  - Does not execute any remaining statements in loop body
- **continue** statement skips the rest of **this** iteration of the loop body
  - Immediately begins the **next** iteration (if there is one)
- **return** statement ends loop and function call
  - May be used with an expression
    - within body of a function that returns a value
  - Or without an expression
    - within body of a function that just does something

Q5-6

File loop pattern

pre-loop computation
for line in file:
    get data from line
    compute using the data
post-loop computation

pre-loop computation
Open file save as fileObject
for line in fileObject:
    process line, which is data as a string
    compute using the data
Close file
post-loop computation

Examine and run the *module6_averageFile.py* module in the project you checked out today.

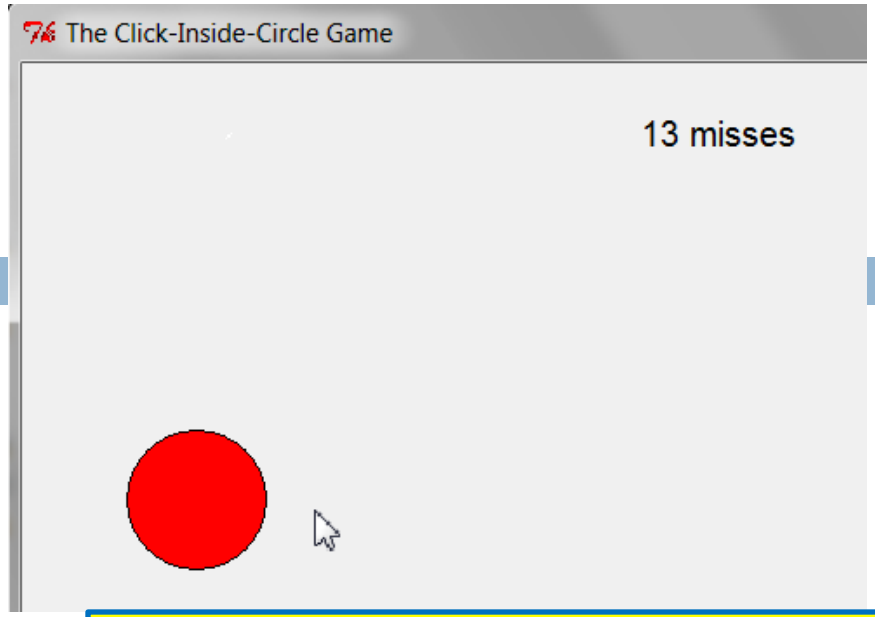
This loop looks like a definite loop but isn't:
it starts reading lines in the file without knowing how many lines it will read before it reaches the end of the file.

Q7

Summary of Loop Patterns

- The compute-in-a-loop pattern
- Six basic compute-in-a-loop patterns:
  - For loop
  - While loop
    - Interactive loop
    - Sentinel loop using a special value as the sentinel
    - Sentinel loop using no-input as the sentinel
  - Loop-and-a-half
    - Combined with use of no-input as the sentinel
  - File loop
  - Nested loops (next session)
  - Wait-for-event loop (next session)

Exercise on Loop Patterns – *Clicks Game*



- In ***module7_clickInsideCircle.py*** you will implement this game:
  - ***The computer shows a circle that jumps around in a window.***
    - *The user chooses how many seconds between jumps and how big the circle is, which correspond to the game's difficulty.*
  - ***The user tries to click inside the moving circle.***
  - *As long as the user misses, the computer displays "XX misses" in the window, where XX is the number of failed clicks so far.*
  - *When the user finally clicks inside the circle, the computer displays "BULLSEYE after XX misses", where XX is the number of failed clicks.*

What loop pattern seems best for this problem?

Answer: The ***sentinel*** pattern, where the sentinel is *any point that is inside the circle*.

Use *classic* or *loop-and-a-half*.

Next: Make main play the game until the user says to stop? What loop pattern for this?

Answer: Again the *sentinel* pattern. What is the sentinel now?

Answer: However the user indicates "stop the game." Here that means any point near the top-left corner of the window.

Exercise on Loop Patterns – *Largest Number in a List of Numbers:* `getList()`

- Implement function `getList()` in module `module8_listMax.py` that:
  - Prompts the user to enter numbers, one at a time
  - Uses a blank line (<ENTER>) as sentinel to terminate input
  - Accumulates the numbers in a list
  - Returns the list of numbers
- Implement the portion of function `main()` that:
  - Tests the above function, by printing the list of numbers entered

What loop pattern seems best for this problem?

Answer: The ***sentinel*** pattern, where the sentinel is *an input that is the empty string*.

You may use either the classic sentinel approach or the loop-and-a-half version of it. Which seems more elegant to you?

Q8, *turn in quiz*

Exercise on Loop Patterns – *Largest Number in a List of Numbers:* `maxList()`

- Implement function `maxList()` in module `module8_listMax.py` that:
  - Takes a list of numbers
  - Returns the maximum (largest) number in the list
- Implement the portion of function `main()` that:
  - Tests the above function, by getting a list from the `getList()` function and printing the maximum (largest) number in the list by calling `maxList()`

What loop pattern seems best for this problem?

Answer: Either of the two FOR loop patterns for looping through a sequence.

Start homework

- When you are through with your individual exercise commit your solutions to your SVN repository
- Start working on homework 13