Checkout today's project
from your individual SVN repository:
13-LoopPatterns

INDEFINITE LOOPS AND LOOP PATTERNS

CSSE 120—Rose Hulman Institute of Technology

Outline

- Return Exam 1 and discuss it
- Debugging
  - And using a Debugger
- Review definite loops
- Indefinite loops
  - while statements
- Loop patterns
- Practice on loop patterns

Comments on Exam 1

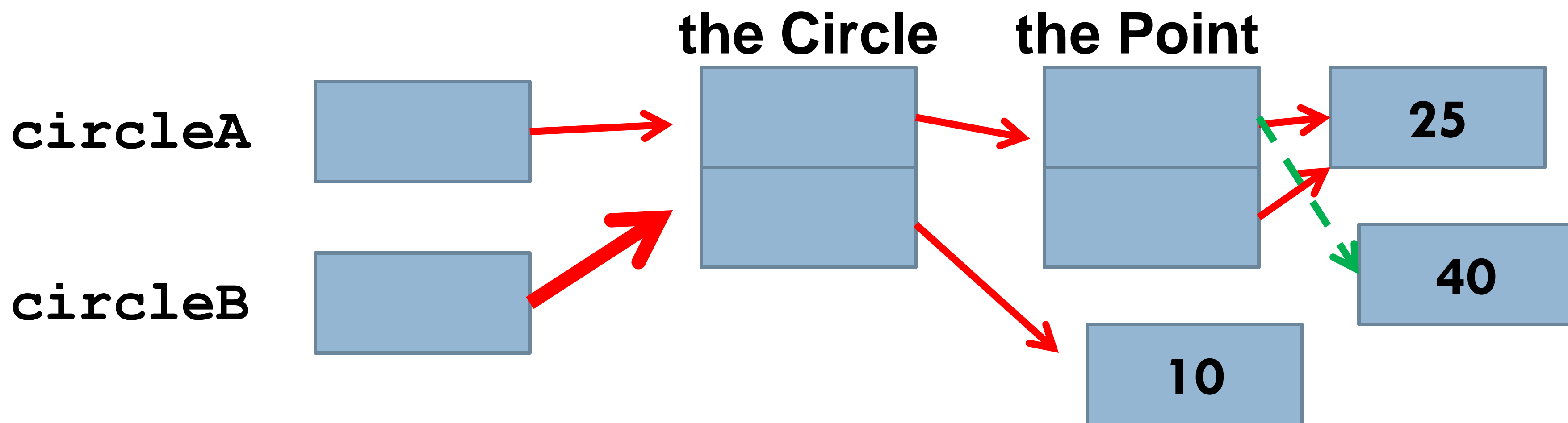
Exception: If your score is 108 (of 120) or higher, no appointment is needed – just tell me in class that you understand what you missed. Be truthful and you thereby earn back 70% of what you missed.

- If you are not satisfied with your score:
  - Set up an appointment with me
  - Review the answer key before your appointment.
    - Find it from the Day 14 resources.
  - At your appointment we will:
    - Make sure you understand the closed-book concepts
    - Together work any of the open-book problems that you missed
    - Select some problems that will give you practice on the concepts which you have not yet mastered
    - Set up a time for you to do some earn-back problems
      - Earn-back problems will be very similar to test problems you missed.
      - Success on the earn-back problems can earn you 70% of the points that you missed. So a C can become an A, and an F can become a B!

Common pitfall on Exam 1 – variables are ***references*** to objects

- What gets printed by the following code:

```
circleA = Circle(Point(25, 25), 10)
circleB = circleA
circleA.move(15, 0)
print circleA.getCenter().getX()
print circleB.getCenter().getX()
```



Common pitfall on Exam 1 – *print* versus *return*

- *print* displays its arguments on the console
- *return* returns its result to the caller, who might or might not print the result
  - E.g., Function A calls function B, who returns x to A.
    Function A then calls function C, who returns y to A.
    Then A computes sin(x) + cos(y)
    and returns that to *its* caller.

All sensible computations, yet nothing is printed by this code fragment. That's normal!

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

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
  - Checkout the **13-LoopPatterns** project from your repository and open its **factorialTable.py** module
  - Run the module. You'll see that it prints wrong numbers for the factorials.
  - Its *factorial* function has two errors (bugs). ***Use the debugger*** (per instructor's demo) to find and fix the errors.

Sample Debugging Session: Eclipse

Run in the debugger

Run to next breakpoint

Single step (step into)

Click this to return to the *Pydev perspective*

A *view* that shows all the executing functions

This is the *Debug perspective*

A *view* that shows all the variables

This *view* is an *editor* that shows the line of code being executed and lets you make changes to the file

A *view* that shows the outline of the module being examined (*Outline View*)

Debug - printFactorial.py - Eclipse SDK

File Edit Source Refactoring Navigate Search Project Run Window Help

Debug | Pydev | Java

Debug

- test printFactorial.py [Python Run]
  - printFactorial.py
    - MainThread
      - printFactorial [printFactorial.py:4]
      - factTable [printFactorial.py:22]
      - <module> [printFactorial.py:24]
      - run [pydevd.py:634]
      - <module> [pydevd.py:779]
  - printFactorial.py

Variables | Breakpoints

Name	Value
Globals	Global variables
formatString	str: %21d
n	int: 0
product	int: 1
width	int: 21

int: 21

printFactorial.py

```
1  def printFactorial(n, width):
2      formatString = "%"+str(width)+ "d"
3      product = 1
4      for i in range(1, n+1):
5          product = product * i
6
7      print formatString % (product)
8
9  #printFactorial(5, 6)
10 #printFactorial(15, 20)
11
12 print "Factorial Table"
13
```

Outline

- printFactorial
- factTable

Console | Tasks

printFactorial.py

```
pydev debugger
Factorial Table
 0
```

Writable | Insert | 4 : 1

Q1

Running a module conditionally

- You can run and/or import modules. If you import a module:
  1. Its methods and other entities become available.
  2. The module is executed.
- Usually you want:
  - Just (1) above if you *import* the module
  - (1) and (2) above if you *run* the module
- Solution:
  - Have a function called (say) *main* that executes whatever the module is intended to execute
  - Put this at the top level of the module:

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

Python sets the special __name__ variable (that's TWO underscores) to __main__ if the module is being run directly (i.e., not as an import).

You'll see this standard boilerplate in most of our forthcoming examples.

Review: Definite Loops

- Review: For loop
  - ***Definite loop***: knows *before the loop starts to execute* the number of iterations of the loop body
    - ***Counted loop***: special case of definite loop where the sequence can be generated by **range()**
  - Example: Most **for** loops
- Syntax:
  - for <var> in <sequence>:
    <body>

Examples of definite loops (first is a counted loop, second is not):

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

sum = 0
for e in list_of_numbers:
    sum = sum + e
```

Indefinite Loops

- Number of iterations is not known when loop starts
- Is a conditional loop
  - Keeps iterating as long as a certain condition remains true
  - Conditions are Boolean expressions
- Typically implemented using **while** statement
- Syntax:

  while <condition> :

  <body>

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

Definite loop

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

Indefinite loop that computes the same sum as the definite loop

Q2-3

While Loop

- A *pre-test loop*
  - Condition is tested at the top of the loop
- Example use of ***while*** loops

  Nadia deposits $100 in a savings account each month. Each month the account earns 0.25% interest on the previous balance. How many months will it take her to accumulate $10,000?

- Open the **moneyDeposit.py** module in your 13-LoopPatterns project.
  - Note the *while* loop.
  - Use the debugger to find the error.

Q4

Exercise on *while* loops

- Open the **findSine.py** module in your 13-LoopPatterns project.
  - Do its two TODO's, using a *while* loop

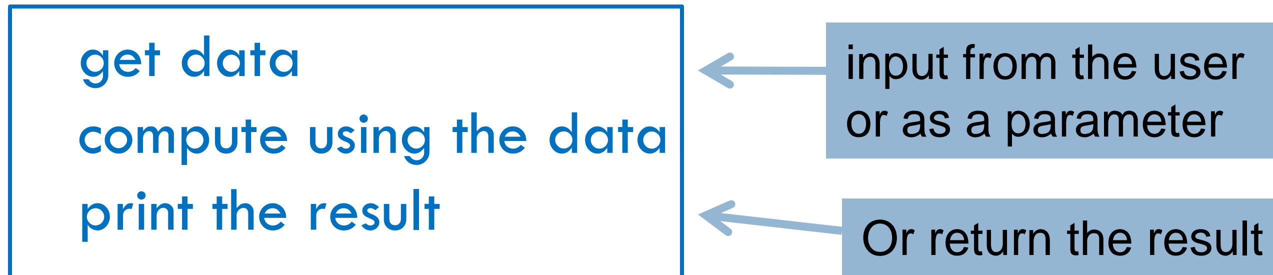
- Questions on the notation for *while* loops?

Outline of Loop Patterns

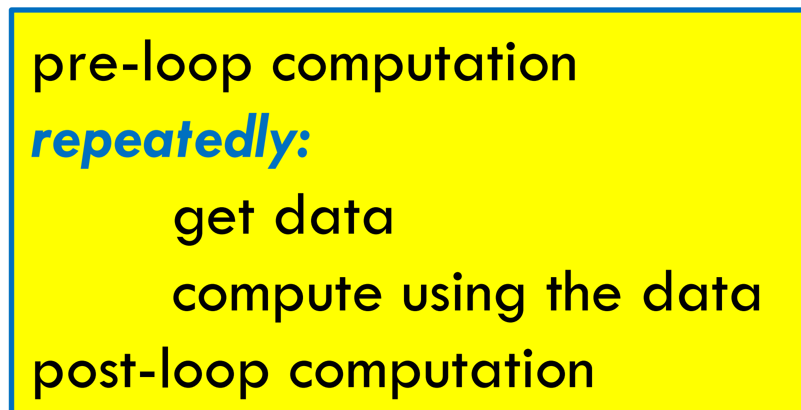
- The compute-in-a-loop pattern
- Six basic compute-in-a-loop patterns:
  - For loop
  - While loop
    - Interactive loop
    - Sentinel loop using impossible values as the sentinel
    - Sentinel loop using no-input as the sentinel
  - Loop-and-a-half
    - Combined with use of no-input as the sentinel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

Loop patterns

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



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



We've seen a special case of this pattern: the ***accumulator*** 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

Q5

For loop pattern

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

- Example: averaging numbers that the user supplies

```
def main():
    ''' Averages a set of numbers that the user supplies. '''
    n = input("How many numbers do you have? ")
    sum = 0.0
    for k in range(n): #@UnusedVariable
        x = input("Enter a number >> ")
        sum = sum + x
    print "\nThe average of the numbers is %0.3f" % (sum / n)
```

Examine and run the ***averageUserCount.py*** module
(part of whose code appears above) in your *13-LoopPatterns* project.

This approach is a lousy way to get numbers that the user supplies. Why?
Answer: user has to count in advance how many numbers she 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 *averageMoreData.py* module in your *13-LoopPatterns* project.

This approach is also a lousy way to get numbers that the user supplies. Why?
Answer: user has to repeatedly answer the “more numbers?” question.

Q6

While loop pattern #2

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

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

```
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 *averageSentinel.py* module in your *13-LoopPatterns* project.

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 it?
Answer: what if you want negative numbers to be included in the average?!

Q7

While loop pattern #3

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

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

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

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

The sentinel value is again not used in calculations.

Examine and run the ***averageOtherSentinel.py*** module in your *13-LoopPatterns* project.

Q8-9

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.

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.

Examine and run the *averageLoopAndAHalf.py* module in your *13-LoopPatterns* project.

Q10-11

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

File loop

- Example:

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

```
def main():
    fileName = raw_input("What file are the numbers in? ")
    infile = open(fileName.strip(), 'r')
    sum = 0.0
    count = 0
    for line in infile:
        sum = sum + eval(line.strip())
        count = count + 1
    print "\nThe average of the numbers is %0.3f" % (sum / count)
```

Examine and run the ***averageFile.py*** module in your *13-LoopPatterns* project.

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.

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 impossible values 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)

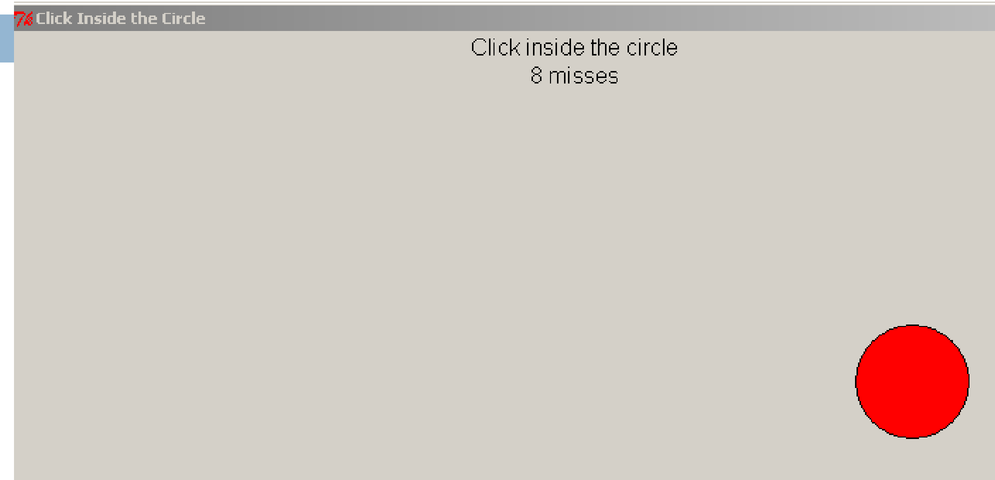
Exercise: While Loops – *guessMyNumber.py*

- In the *guessMyNumber.py* module in your *13-LoopPatterns* project, you will implement the following game:
  - *The computer generates a random integer between 1 and 100. The user then keeps entering guesses for that number until he or she guesses correctly. For each incorrect guess, the computer tells whether the guess was too high or too low and asks again. Once the correct guess is made, the computer congratulates the user and prints the number of guesses made.*

- What loop pattern seems best for this problem?
  - Consider using the sentinel pattern, where the sentinel is the secret number.

Q14

Exercise: While Loops – ***clickInsideCircle.py***



- In the ***clickInsideCircle.py*** module in your *13-LoopPatterns* project, you will implement the following game:
  - *The computer shows a circle that jumps around in a window. (The user chooses how many seconds between jumps, which corresponds 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?
  - Consider using the sentinel pattern, where the sentinel is any point that is inside the circle.

Start homework

- Start working on homework 13
  - The preceding exercises are part of it.