CS 115 Lecture 11

The debugger; while loops

Neil Moore

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

15 October 2015
20 October 2015

Debugging

We've seen how to write test cases to help you find bugs in your program. What to do when you find that something is wrong?

- Find the line that's wrong, and fix it.
  - Which line is wrong?
  - We know the symptoms, not the disease.
  - So our job that of a doctor...
  - Find out what's making the patient sick!
- Sometimes just reading and tracing the program is enough.
  - And once upon a time that was the only option.
  - But doctors can run tests, ask questions, ...
  - If we could interact with the program while it's running...

    ...then we might be able to see the bug as it happens.

The debugger

The **debugger** is a tool for controlling and inspecting a program as it runs.

- Part of most IDEs (WingIDE and IDLE included).
- It *doesn't* find the bugs for you.
- Instead, it:
  - "Slows down" the program by letting you run it one line at a time.
  - Shows you variables and their values.
  - Shows what functions are currently running ("call stack").

Breakpoints

When using the debugger, you usually start by setting a **breakpoint**.

- This tells the debugger to pause the program just before it executes that line.
- In WingIDE, click next to the line number.
  - A red dot appears to show there is a breakpoint there.
  - Click again to turn them off.
  - Note: breakpoints are **not** saved with the program!
- In IDLE, right click the code and "Set breakpoint".
  - The line will be highlighted in yellow.
- To run under the debugger, click the "Debug" icon in Wing.
  - In IDLE, select **Debug** → **Debugger** then run the program normally.
  - It will run the program full-speed until it reaches the breakpoint.
  - Then it pauses the program and gives you control.
  - If you run the program without "Debug", breakpoints are ignored!
- With branches and loops, you might need several breakpoints to make sure execution reaches one of them.

Single-stepping

Single stepping means running the next line of the program. In most IDEs there are three ways to single-step.

- Step Over ("Over" in IDLE)
  - Execute the current line, pausing again when it finishes.
  - If the line calls a function, will execute the whole function.
- Step Into ("Step" in IDLE)
  - Execute the current line, pausing at the next line executed.
  - If the line calls a function, will pause at the beginning of that function.
- Step Out ("Out" in IDLE)
  - Execute the rest of this function, pausing after it returns.
  - Stepped into a function you didn't want to see? Use this to get back.
- More about the differences when we cover functions.
- These only work when your program is paused by the debugger.
  - So only in debug mode.
  - And you need a breakpoint first!

Debugging variables

The **watch window** shows you the variables that exist before executing the current line.

- In WingIDE, called "Stack Data".
- In IDLE, called "Locals".
- Shows variables that are **in scope**:
  - Already defined in this function.
  - Not destroyed yet.
- Also shows their values and types.
- Watch this window when single-stepping.
  - That will show you what variables are changing and how.
  - Make sure they change as you expect!

Debugging example

Let's see an example of the debugger in action.

- `triangular-bug.py`

While loops

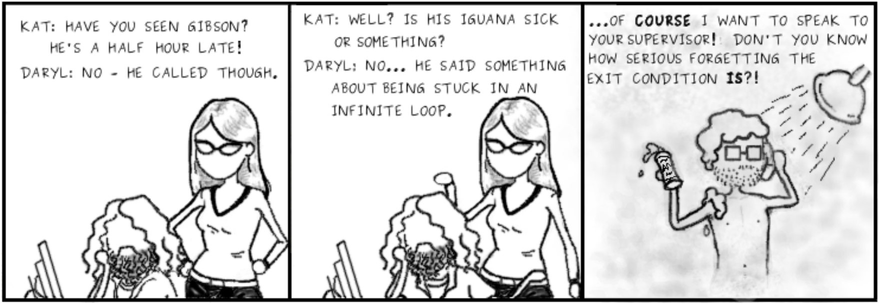


Image: `skunklogic.com`, 2011

How do we fix this?

1. Lather
2. Rinse
3. Repeat if necessary

Bounded and unbounded iteration

The for loops we've been using are **bounded** or **definite** loops.

- The interpreter decides in advance how many times it will run.
- That doesn't change once the loop starts.

  ```
  for i in range(5):
      i = 1
      print(i)
  ```

  Even though we're changing *i*, the loop still runs five times!
- That's kind of nice because it means no infinite loops.
- But. . .

Unbounded iteration

- In 1928 (long before computers existed!), German mathematician Wilhelm Ackermann proved that bounded iteration is not sufficient to define all mathematical functions.
  - In mathematician-speak he called it "primitive recursion".
  - Wikipedia "Ackermann function" if you're curious.
- A simpler example: asking the user to input a positive number.
  - If they give a negative number or zero, we'll repeat the question.
  - How many times do we need to repeat?
  - No way to know in advance.
- So we need some way to make a loop that can run an unlimited number of times.
  - An **unbounded** or **indefinite** loop.
  - Indefinite, not infinite!
    - Most loops should still stop eventually—we just can't predict when.
    - "Repeat if necessary." — We just need to say when it's necessary.

While loop syntax and semantics

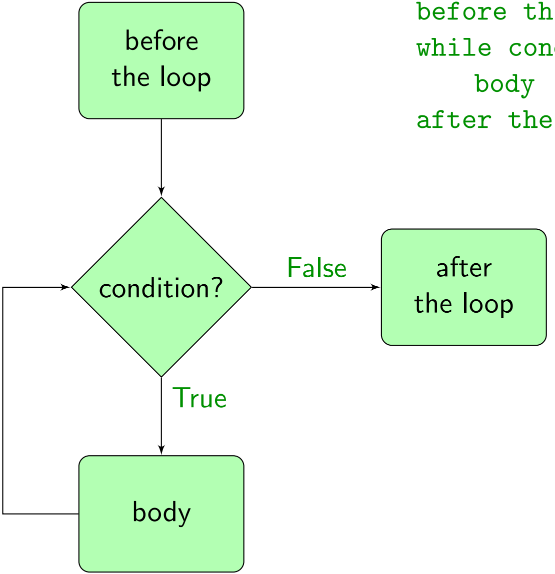
- Syntax:

```
while condition:
    body
```

  - The condition is a boolean expression, just like `if`.
  - The body is... an indented block of code.
    - ★ (How many times have you heard that?)
- Semantics:
  1. Check the loop condition.
  2. If false, end the loop.
  3. If true, run the body and repeat.

  - If the condition was false to begin with, the body never runs!
- Structured programming guarantees:
  - One entrance: always starts by checking the condition.
  - One exit: ends only when the condition is false.
  - Only checks the condition again after running the entire body.

Flowchart for `while`




```
before the loop
while condition:
    body
after the loop
```

Common errors

- What happens if the condition is a tautology (always true)?
  - The loop never ends!
  - An **infinite loop** is usually an error.
  - If you really do want to run forever:

    ```
    while True:
        ...
    ```
- Conversely, what if the condition is a falsehood?
  - Then the body will never run at all.
  - What do we call code that never runs?
    - Dead code!

Sentinel logic

One common use for a while loop is to process data.

- We want to keep getting more data until we see a special value.
  - Read from a file until you see a blank line.
  - Ask for a number until they enter 0.
  - Get commands from the user until they type "quit".
  - Read and log temperatures until one is out of range.
  - This special value (or values) is called a **sentinel**.
    - Just a fancy word for "guard".
- Sounds like a job for a while loop!
  - But a while loop checks at the very beginning.
  - But we can't check until we have the first input!

Loops with sentinel logic

The solution: the sentinel loop pattern.
Get the input in two places:

- Just before the loop.
- As the last step of the loop body.

1. Get the first input
2. While the input is not a sentinel:
    2.1. Do something with the input.
    2.2. Get the next input.

```
num = int(input("Enter a number, 0 to exit: "))
while num != 0:
    print("Its reciprocal is", 1 / num)
    num = int(input("Enter another number, 0 to exit: "))
```

Input validation

Sentinel logic can also be used for input validation.

- That is, asking the user until we get a "good" value.
- We just need a slight twist:
  - We repeat the loop if the input is *bad*.
  - So our "sentinel" is *any good input*.
  - What goes in the loop body?
    - An error message.
    - Then get the next input.

Example: `positive.py`

Loop example: numeric input

Remember our programs with `int(input(...))`. What happened when the user provided non-numeric input?

- It crashed. . . what part crashed, exactly?
  - The type cast!
  - So can we validate the input before doing the type cast?
- Python gives us a way to check if a string contains only digits.
  - `mystr.isdigit()` — What does the dot syntax mean?
    - Calling a method: Strings are objects!
  - `isdigit` returns true if:
    - The string contains no non-digit characters; *and*
    - It has at least one digit (so the method is badly named).
  - Only works for non-negative integers! "." and "-" are not digits!
    - There are other special cases, too (Arabic digits, . . .)
    - In a later class, we'll see how to do this properly with exceptions.
- `numeric.py`

Loop example: Collatz conjecture

Consider the following arithmetic procedure on an integer.

- If the integer is even, divide it by 2.
- Otherwise, multiply by 3 and add 1.
- What will happen if we repeat this process?
- $1 \to 4 \to 2 \to 1 \to \cdots$
- $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1 \cdots$
- Will we always get back to one?
  - Lothar Collatz (1937) conjectured that we always get back to 1.
  - But nobody knows for sure!
  - Computers have tested all numbers less than 5.764×10^{18}
  - So far they all get back to 1 eventually, but there's no proof that bigger numbers always will.
- Let's write a program to test it!
  - `collatz.py`

Flags and while loops

Sometime we only discover inside the loop that we want to leave.

- For example, "Do you want to play again?"
- Python lets us `break` out of the loop...
  - ...but that violates the structured programming constraints.
  - So we won't use it in CS 115.
- Instead, we can set a **flag**.
  - A boolean variable (we saw these with `if`).
  - Use the flag as the condition of the `while`.
  - Initialize the flag before the loop:
  - When we discover that we're done, change the flag.

While loop with a flag

```
done = False
while not done:
    play a round
    answer = play_again()
    if not answer:
        done = True
```

Why don't we use sentinel logic?

- Sentinel logic would ask before the loop...
- But we want to play one round before we ask!

While loops with a counter

While loops can be combined with a counter to simulate a for loop over a range.

- Initialize the counter (to the start).
- While the counter is less than the stop:
  - Do stuff.
  - Add the step to the counter.

```
i = 0
while i < 4:
    print(i, "squared is", i**2)
    i += 1
```

- Why would you ever write it this way?
  - Maybe the step size changes.
  - Maybe there are multiple stopping conditions.

Another loop example: any

Remember our "any" for loop pattern:

```
anyodd = False
for num in 2, 1, 4, 16, 12, 0, 3:
    if num % 2 == 1:
        anyodd = True
```

- What if we don't have all the numbers in advance?
  - Then we can't use a `for` loop.
- Also, notice it keeps going even when we're sure the answer is True.
  - (because `anyodd` can never become False again)
- As soon as we see an odd number, we can stop, because the answer must be True!
- We can solve both problems with a while loop.

While loop example: any

Let's solve the first problem and make it accept any number of inputs.
We need a sentinel value: let's use 0.

```
anyodd = False
num = int(input("Enter an integer, 0 to quit."))
while num != 0:
    if num % 2 == 1:
        anyodd = True
    num = int(input("Enter an integer, 0 to quit."))

if anyodd:
    print("At least one number was odd.")
else:
    print("No odd numbers found.")          anyodd.py
```

It still asks for more numbers after an odd one. Let's fix that.

While loop with multiple exits

- Stop the loop if `anyodd` is true.
  - (It can never become false again).
- But we still want to stop the loop if the input is zero
  - Stop if the loop if `anyodd` is true **or** if the input is 0.
  - Continue the loop if anyodd is false **and** the input is not zero.
    - ⋆ **de Morgan's law**: "not (A or B)" = "(not A) and (not B)"

While loop example: any

```
anyodd = False
num = int(input("Enter an integer, 0 to quit."))
while not anyodd and num != 0:
    if num % 2 == 1:
        anyodd = True
    else:  # don't ask if we're just going to quit
        num = int(input("Enter an integer, 0 to quit."))
if anyodd:
    print("At least one number was odd.")
else:
    print("No odd numbers found.")
```

anyodd-2.py

Common errors with flags

- Accidentally using `or` instead of `and`.
  - The loop will run too long.
  - If the condition is a tautology, infinite loop.
- Accidentally using `and` instead of `or`.
  - The loop won't run long enough.
  - If the condition is a falsehood, doesn't run at all.
- Not initializing the flag.
  - Run-time error.
- Always setting the flag.
  - Let's see an example of that.

Broken loop: unconditional flag

```
anyodd = False
num = int(input("Enter an integer, 0 to quit."))
while num != 0:
    if num % 2 == 1:
        anyodd = True
    else:
        anyodd = False
    num = int(input("Enter an integer, 0 to quit."))

if anyodd:
    print("At least one number was odd.")
else:
    print("No odd numbers found.")
```

- This version sets the flag every time through the loop.
- So the flag won't actually mean "are any numbers odd?".
- Instead, this version only cares about the *last* number.
  - Seeing an even number shouldn't change our answer at all!

Sentinel logic: recap

To recap sentinel logic:

- Repeating a loop until you see a special value.
- Get input both **before** the loop,
  and **at the end** of the body.
- "Enter a number, or 0 to quit."
  - Keep asking until we see the **sentinel** 0.
  - While the number is not zero...
  - Inside the loop, process the (nonzero) number.
- In reverse: input validation.
  - Keep asking until we get what we want.
  - Now a *good* value makes us leave the loop.
  - While the number is not positive...
  - Inside the loop, print an error message.
  - Process a valid input after the loop is over.