CS 115 Lecture 10

Structured programming; `for` loops

Neil Moore

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

8 October 2015
13 October 2015

The bad old days: GOTO

In the early days of programming, we didn't have for loops, if statements, etc.

- Instead, we had simply "if this is true, go to line 10".
- Could use that to skip over code (like an **if**).
- . . . or go to an earlier line to write a loop.
- This was very tedious and error prone.
  - . . . especially if something has to be changed.
  - "Spaghetti code": trying to trace a program was like trying to trace one strand in a plate of spaghetti.

Structured programming

- In the 1960s, computer scientists started to think about how to write programs that were easier to understand and follow.
  - Edsger Dijkstra, "Go To Statement Considered Harmful" (1968).
- They introduced the paradigm of **structured programming**.
  - Patterns that lead to easier-to-understand programs.
    - Easier to test and debug.
    - Easier to modify and maintain.
    - Easier to collaborate on large programs.

Data structures and control structures

- We've already seen a little about **data structures**:
  - Ways of organizing data within a program.
    - (Remember, in the computer it's all binary)
  - Simple: Constants, variables.
  - More complex: Graphics objects, strings, lists...
- **Control structures** are ways of controlling the execution of a program.
  - Which statements execute, and in which order.

The three basic control structures

In 1966, Böhm and Jacopini showed that any program using “go to” could be rearranged to use only three simple control structures.

- Sequence.
- Selection.
- Iteration.
- We'll add a fourth: **Subprograms** (more in chapter 5).

Each of these control structures has two important guarantees:

- Only one way to enter the control structure.
- Only one way to leave the control structure.
- *One entrance, one exit.*

Sequence

"Sequencing" or "sequential execution" just means:

Running one statement after another.

- In Python we just write one line after the next.
- "The default" in some sense.
- Guarantees:
  - The steps will execute in the order given.
  - Steps will not be skipped.
  - Will always start at the first statement of the sequence...
  - and finish at the last statement.

Selection

"Selection" means choosing which code to run based on some condition or question.

- In Python, an **if-else** statement.
- Two branches: true and false.
  - Each branch is another control structure (most often a sequence).
- Guarantees:
  - Always starts with the question/condition.
  - Runs one branch or the other, never both.
  - ...and never neither.
- Avoid **dead code**: code that is never executed.
  - Often because the condition is always true or always false.

Iteration

"Iteration" means running code multiple times (a loop).

- In structured programming, "repeat this body until a condition is false".
- In Python, a **while** loop (in about a week).
  - **for** loops are a special case of iteration.
- Guarantees:
  - Always starts with the question/condition.
  - If the condition is true, executes the **entire** body, then comes back to the condition.
  - Otherwise (the condition is false), leaves the loop.
- Be careful to avoid **infinite loops**, where the condition is always true.

Subprograms

Sometimes we may need to repeat the same combination of control structures in several different places.

- It would be nice if we didn't have to write the code multiple times.
- A **subprogram** is a chunk of the flowchart treated as a single unit.
- When we need to execute those steps, we **call** the subprogram.
  - Run the subprogram, wait for it to finish.
  - Keep going where you left off.
  - Sometimes we send values to the subprogram.
  - And sometimes the subprogram sends a value back.
- In Python, subprograms are called **functions**.
  - **Arguments** are the values we send to the subprogram.
  - And the function can **return** a result.
  - Can you think of Python functions that:
    - Take one or more arguments?
    - Take no arguments?
    - Return a result?
    - Don't return a result?

Control structures summary

- **Sequence** (one statement after the other: easy to forget) ✓
- **Selection** (conditionals: `if`) ✓
- **Iteration** (loops: `for` and `while`)
- **Subprograms** (functions: `def`)

We've seen sequence and selection already, so now let's look at iteration in more detail.

Repeating yourself

What if we wanted to draw a tic-tac-toe board with 4×4 lines?

- We could write code to draw a vertical line...
- ...and code to draw a horizontal line...
- We need to do that four times each.
  - With different coordinates each time.
- Do we have to copy-and-paste each one 4 times?
  - Of course not!
  - **Loops** allow you to execute code multiple times.
    ...with a variable that is different each time.
- Two kinds of loop: definite and indefinite.
  - **Definite loops** know in advance how many times to run.
  - **Indefinite loops** run until some condition is satisfied.
  - Today we'll see how to write definite loops in Python.

The **for** loop

- Syntax: `for` *`var`* `in` *`sequence`* `:`
  - Followed by a **block** (collection of indented lines) called the **body**.
    - ★ The body must be indented past the "for"!
  - *`var`* is an identifier (variable name).
- Semantics: Execute the body once for each item in the sequence.
  - Each time, the variable `var` will have the value of that item.
  - Each run of the body is called an **iteration**.
- A very simple for loop:

```
for color in ('red', 'green', 'blue'):
    print(color, 'is a primary color.')
```

- We're giving a **tuple**, but a list in square brackets would work too.
- When executed it does:

```
Iteration 1: print('red', 'is a primary color.')
Iteration 2: print('green', 'is a primary color.')
Iteration 3: print('blue', 'is a primary color.')
```

Other kinds of sequences

Strings can also be used as sequences. Each iteration of the loop operates on a single character:

```
name = input("What is your name?  ")
for char in name:
    print(char)
```

- Prints this:

```
M
o
o
r
e
```

Numeric ranges

One of the most common, and most useful, kinds of sequence for a `for` loop is a numeric range. In Python, you create numeric ranges with the `range` function. There are three ways to call `range`:

- `range(3)`: counts from 0 up to 2.
  - Computer scientists usually count from zero, not one.
  - Goes up to but *not including* the number. (just like `randrange`!)

    ```
    for i in range(3):
        print(i, "squared is", i**2)
    ```

    Prints:

    ```
        0 squared is 0
        1 squared is 1
        2 squared is 4
    ```
  - Notice the loop ran 3 times (0, 1, 2).
    - ⋆ Don't make a fencepost error!

More ranges

We can also tell `range` to start at a different number:

- Syntax: `range(start, stop)`
  - Produces a sequence of integers from start to stop.
  - Does include the start (inclusive), not the stop (exclusive).

    ```
    for i in range(3, 6):
        print(i)
    ```

    Prints:

    ```
        3
        4
        5
    ```
  - Runs for `(stop - start)` iterations.
- What if we wrote `range(1, 1)`?
  - Empty sequence: stops before getting to 1.
  - The loop wouldn't run at all! **Loops can run for 0 iterations.**
  - Similarly, `range(5, 1)` is an empty sequence.
    - ★ So this loop will do nothing:

      ```
      for i in range(1, 5, -1):
          print(i)
      ```
    - ★ The body never executes (is **dead code**).

Counting with steps

Finally, we can tell `range` to count by steps, only considering every *n*th number:

- Syntax: `range(start, stop, step)`
  - Instead of adding 1 in each iteration, adds *step*.
  - The first number is still `start`.
  - The next number is `start + step`, then `start + 2*step`, ...
  - What will this do?

    ```
    for i in range(10, 25, 5):
        print(i)
    ```
  - Prints:

    ```
    10
    15
    20
    ```
  - Does not include 25: stop is still exclusive.
- What about `range(10, 2)`?
  - Two arguments are start and stop, *not* step.

Counting backwards

You can count down by providing a negative step.

```
for i in range(3, 0, -1):
    print("Counting down:", i)
print("Lift off!")
```

- Prints:

```
Counting down: 3
Counting down: 2
Counting down: 1
Lift off!
```

- The stop is still exclusive.
- `range(1, 5, -1)` is an empty sequence.

Tic-tac-toe grid

- Now we can make that tic-tac-toe grid.
- We'll have one loop to draw the vertical lines.
- And another to draw the horizontal lines.
- `grid.py`
- A neat "display hack" (simple code to make an intricate picture) using for loops and if:
  `moire.py`

Averages

Suppose we have a collections of measurements in a list, and we want to find their average: add them all up and divide by the number of measurements.

```
temperatures = [67.0, 69.2, 55.3, 71.2, 65.4]
```

- We can get the length with `len(temperatures)`
- For the sum, we need some kind of loop.

  ```
  for temp in temperatures:
  ```
- We'd need to add the next number in each iteration.
- We need a variable to keep track of the sum.
  - We call such a variable an **accumulator**.
- Accumulators aren't new syntax.
  - Just a new way of using assignment.
  - A **logical** concept, used in most programming languages.

Accumulators

The general idea of accumulators:

- Make an accumulator variable to hold the "total".
  - Like the display on a calculator.
- Before the loop, **initialize** it to a known value.
  - Clear the calculator first!
  - If we are calculating a sum, start at 0.

    ```
    total = 0
    ```
    - 0 is the **identity** for addition: Adding 0 to a number doesn't change it.
- Inside the loop, use assignment to update the accumulator.

  ```
  for temp in temperatures:
      total = total + temp
  ```
  - Or use augmented assignment:

    ```
        total += temp
    ```
- What if we didn't initialize `total` first?
  - `NameError: name 'total' is not defined`

Accumulators

Accumulators can be used for more than just addition.

- Choose the initial value carefully so it doesn't change the result.
- **Factorial**: $1, 2 = (1 \times 2), 6 = (1 \times 2 \times 3), \ldots$
  - Inside the loop we will multiply the accumulator.
  - If we started with 0, we'd never get anything but 0.
  - The multiplicative identity is 1: use that.

```
factorial = 1
for i in range(1, max + 1):
    factorial *= i
```

- Counting: how many times does something happen?
  - Just like sum: initialize with 0.
  - Instead of adding *i*, just add 1.

```
numodd = 0
for i in range(1, 100, 2):
    numodd += 1
```

  - We call an accumulator like this a **counter**.

More accumulators

- Reversing a string.
  - Our accumulator will be a string.
  - We'll loop over the characters of the input string.
  - Concatenate each new character to the *beginning* of the accumulator.
    - ★ What is the identity for concatenation?
    - ★ (What can you concatenate with without changing the answer?)
    - ★ The empty string!

```
instr = input("Enter a string:  ")
reversed = ""
for char in instr:
    reversed = char + reversed
print(instr, "backwards is", reversed)
```

- `reverse.py`

Previous-current loop

Sometimes a loop needs two items from the sequence at once.

- Drawing lines, computing distances.
- Or to see if user input has changed.
- We can save the "previous" item in a variable.
  1. Initialize `prev`
  2. Loop:
     1. `curr` = the new item.
     2. Do something with `prev` and `curr`.
     3. `prev = curr`
- In the first iteration, `prev` is the initial value.
- On following iterations, `prev` is the value from the preceding iteration.

Tracing code

- Code with loops, several values, etc. can get complicated.
- It's good to know what it will do before running it.
  - Trial and error is good for practice and experimentation.
  - Not so good for making working, bug-free code.
- We'll learn several debugging techniques in class.
  - One of the simplest and most useful is **tracing**.
    - Also known as a "desk check".
  - Run though code line-by-line, simulating its behavior.
  - Keep track of the variables and output.
  - *Pretend you are the interpreter*

Tracing a previous-current loop

```
1: prev = get mouse
2: for i in range(2):
3:     curr = get mouse
4:     draw line from prev to curr
5:     prev = curr
```

Line	i	prev	curr	output
1	–	(50, 50)	–	
2	0	(50, 50)	–	
3	0	(50, 50)	(400, 50)	
4	0	(50, 50)	(400, 50)	One line
5	0	(400, 50)	(400, 50)	
2	1	(400, 50)	(400, 50)	
3	1	(400, 50)	(200, 300)	
4	1	(400, 50)	(200, 300)	Another line
5	1	(200, 300)	(200, 300)	

Drawing program

Let's write a program that lets the user click on a sequence of points to draw a path.

- What do we need to draw a line?
  - Two points.
  - The previous point, and the new one.
- We'll have a loop where the user clicks on points.
  - Draw a line from the previous point to the new one.
  - No line for the first point.

Flag variables

A **flag** is another word for a boolean variable.

- Often used with a loop, like an accumulator.
  - Set the flag to `True` or `False` before the loop.
  - Inside the loop, maybe set it to the opposite.
  - After the loop, check the flag's value.

Common patterns: any

- As an example of a flag variable, let's check whether any of a sequence of numbers is negative.
- We'll start with a flag.

```
any_neg = False  # None so far...
for number in 0, 5, 12, -1, 2:
    if number < 0:
        any_neg = True  # Found one!
if any_neg:  # Or if any_neg == True:
    print("Some number was negative")
```

- To check "some" or "any":
  - Initialize the flag to `False`.
  - Set it to `True` if you find something.

Common patterns: all

- Checking if something is true for **all** inputs is the opposite of “any”:
  - Initialize the flag to `True`.
  - Set it to `False` if you find an exception.

```
all_even = True  # No exception yet
for number in 8, 12, 2, 1:
    if number % 2 != 0:  # if not even
        all_even = False
if all_even:
    print("Every number was even.")
```

- Remember, you must initialize the flag before the loop!

Adding some features

Let's add two features to our program:

1. We'll ask the user for the number of points.
   - Using an `Entry` object.
2. We'll count and display the total length of the lines.
   - Using an accumulator in the loop.
   - And a `Text` object to display the length.
   - Distance formula: $dist = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Testing loops

How to test a loop?

- Verify that it runs the correct number of times.
- What if the number of iterations is controlled by the user?
  - For example, our drawing program.
  - What situations might cause an error?
    - The code might fail when the loop doesn't run.
    - Or it might fail on the first iteration.
    - Or it might only fail with multiple iterations.
  - So you need three test cases:
    1. The loop doesn't run at all.
    2. The loop runs once.
    3. The loop runs several times.
- The three most important numbers in CS: 0, 1, many.