CS 115 Lecture

For loops

Taken from notes by Dr. Neil Moore

Repeating yourself

What if we wanted to play four rounds of a game?

- We could write code to play one round, but
- We need to do that four times each
  - with different inputs and results each time.
- Do we need to copy-and-paste the code 4 times?

Repeating yourself

- Do we need to copy-and-paste the code 4 times?
  - No!
  - **Loops** allow you to execute code multiple times.
    with a variable that is different each time
- Two kinds of loops: definite and indefinite
  - **Definite loops** know in advance now 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 more than the “for” line!
  - *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.**

The **for** loop

- A very simple for loop:

```
for color in (‘red’, ‘green’, ‘blue’):
    print(color, “is a primary color.”)
```

- We give a **tuple** but a list in square brackets would work too
- When executed, it does:

Iteration 1: `red is a primary color`

Iteration 2: `green is a primary color`

Iteration 3: `blue is a primary color`

Other kinds of sequences

Strings can 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:

```
J
o
h
n
```

Numeric ranges

One of the most common, and most useful, kinds of sequences for a `for` loop is a numeric range. In Python, you create numeric ranges with the `range` function. It always creates integers. 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 final 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 2
```

  - Note the loop ran 3 times (for i = 0, 1, and 2)
    - Don't make a fencepost error!

Range variations

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

- Syntax: `range(start, stop)`
  - Produces a sequence of integers from start to stop
  - Includes the start number (inclusive), does NOT include the stop number (exclusive)

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

prints:

```
3
4
5
```

  - Runs for `(stop - start)` iterations

Variations on range

- What if we wrote `range(1, 1)`?
  - It gives an empty sequence: stops before getting to 1
  - The loop won't run at all! **Loops can run for 0 iterations!**
  - Similarly, `range(5, 1)` is an empty sequence

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

    - The body never executes (is **dead code**)

Counting with steps

The last variation on range: We can count by steps bigger than 1, only considering every *nth* 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`, …

Counting with steps

- What will this do?

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

  - Prints:

```
10
15
20
```

  - Does not include 25, the stop number is still exclusive.
- What about `range(10, 2)` ? # common error!
  - Since there are only two arguments, it means start at 10 and stop at 2, NOT start at 0, stop at 10 and step 2!

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 number is still exclusive (not included)!
- `range(1, 5, -1)` is an empty sequence

Finding an average

Suppose we have a collection 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 number of measurements by a function called len: len(temperatures)
- For the sum, we need some kind of a loop

  ```
  for temp in temperatures:
  ```

- We need to add another number in each iteration
- We need a variable to keep track of the sum
  - We call such a variable an **accumulator**
- Accumulators are NOT new syntax
  - Just a new way of using assignment
  - A **logical** concept, used in most programming languages

Accumulators

The general pattern of accumulators:

- Make an accumulator variable to hold the “total”
  - Like the display on a calculator
- Before the loop starts, **initialize** the accumulator to a known value
  - Like clearing out 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

Accumulators

- 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 don't initialize total first?
  - `NameError: name 'total' is not defined`

Accumulators

Accumulators can be used for more than just adding bunches of numbers.

- Choose the initial value carefully so it doesn't change the result
- **Factorial:** 1, 2 = (1 x 2), 6 = (1 x 2 x 3), …
  - Inside the loop we will multiply the accumulator
  - If we started the accumulator at zero, we'd never get anything but zero!

Accumulators

- 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 zero.
  - Instead of adding *I*, just add 1.

```
numoff = 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 string
    - What is the identity element for concatenation?
    - (That is, what can you concatenate with, without changing the original string?)
    - The empty string!

Reversing a string

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

See reverse.py

Previous-current loop

Sometimes a loop needs two items from a sequence at one time

- Drawing lines (needs 2 points at once), 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 variables, 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 through code line-by-line, simulating its behavior
  - Keep track of the variables (RAM) and output
  - *Pretend you are the interpreter* and you are NOT SMART!

Are the data in ascending order?

- Example of prev/curr pattern
- You need to compare two pieces of data at a time

```
in_order = True
prev = int(input("enter some data "))
for i in range(5):
    curr = int(input("Enter some data "))
    if prev > curr:
        in_order = False
if in_order:
    print("all were in order")
else:
    print("at least one pair was out of order")
```

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)	