CSc 110, Spring 2017

Lecture 11: `while` Loops, Fencepost Loops, and Sentinel Loops

Adapted from slides by Marty Stepp and Stuart Reges

Strings and ACSII values (decimal)

- All characters are assigned numbers internally by the computer, called *ASCII* values.

  - Examples:
    `'A'` is 65, `'B'` is 66, `' '` is 32
    `'a'` is 97, `'b'` is 98, `'*'` is 42

  - We can get the ASCII value of a `String` of length 1 using `ord(str)`
    `ord('a')` is 97

  - The function `chr(n)` returns the character represented by the ASCII value n
    `chr(66)` is 'B'

  - This is useful because you can do the following:
    `chr(ord('a') + 2)` is `'c'`

String question

- A *Caesar cipher* is a simple encryption where a message is encoded by shifting each letter by a given amount.
  - e.g. with a shift of 3, A → D, H → K, X → A, and Z → C

- Write a program that reads a message from the user and performs a Caesar cipher on its letters:

```
Your secret message: Brad thinks Angelina is cute
Your secret key: 3
The encoded message: eudg wklqnv dqjholqd lv fxwh
```

Fencepost loops

A deceptive problem…

- Write a method `print_letters` that prints each letter from a word separated by commas.

  For example, the call:

  ```
  print_letters("Atmosphere")
  ```

  should print:

  ```
  A, t, m, o, s, p, h, e, r, e
  ```

Flawed solutions

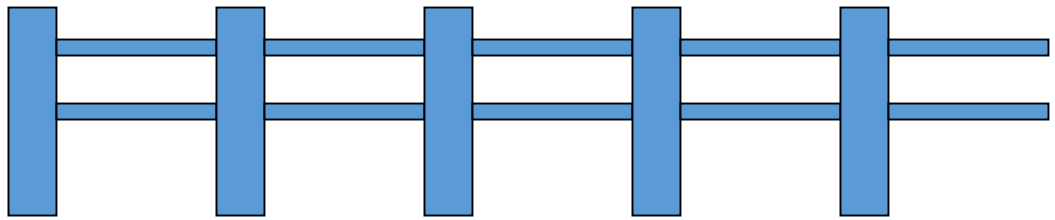
- ```
  def print_letters(word):
      for i in range(0, len(word)):
          print(str(word[i]) + ", ", end="")
      print()    # end line
  ```
  - Output: `A, t, m, o, s, p, h, e, r, e,`

- ```
  def print_letters(word):
      for i in range(0, len(word)):
          print(", " + str(word[i]), end="")
      print()    # end line
  ```
  - Output: `, A, t, m, o, s, p, h, e, r, e`

Fence post analogy

- We print *n* letters but need only *n* - 1 commas.
- Similar to building a fence with wires separated by posts:
  - If we use a flawed algorithm that repeatedly places a post + wire, the last post will have an extra dangling wire.

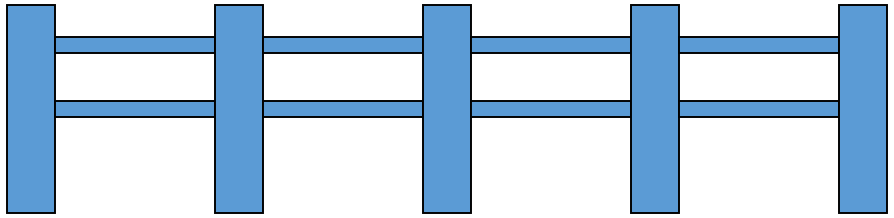
for length of fence :
    place a post.
    place some wire.



Fencepost loop

- Add a statement outside the loop to place the initial "post."
  - Also called a *fencepost loop* or a "loop-and-a-half" solution.

place a post.
for length of fence ***– 1****:*
    place some wire.
    place a post.



Fencepost method solution

- ```
  def print_letters(word):
      print(word[0])
      for i in range(1, len(word)):
          print(", " + word[i], end="")
      print()   # end line
  ```

- Alternate solution: Either first or last "post" can be taken out:

```
def print_letters(word):
    for i in range(0, len(word) - 1):
        print(word[i], end=", ")
    last = len(word) - 1
    print(word[last]) # end line
```
```

Fencepost question

- Write a function `print_primes` that prints all *prime* numbers up to a max.
  - Example: `print_primes(50)` prints
    `2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47`
  - If the maximum is less than 2, print no output.

- To help you, write a function `count_factors` which returns the number of factors of a given integer.
  - `count_factors(20)` returns `6` due to factors 1, 2, 4, 5, 10, 20.

Fencepost answer

```
# Prints all prime numbers up to the given max.
def print_primes(max):
    if (max >= 2):
        print("2", end="")
        for i in range(3, max + 1):
            if (count_factors(i) == 2):
                print(", "  + str(i))
        print()

# Returns how many factors the given number has.
def count_factors(number):
    count = 0
    for i in range(1, number + 1):
        if (number % i == 0):
            count = count + 1   # i is a factor of number
    return count
```

`while` loops

Categories of loops

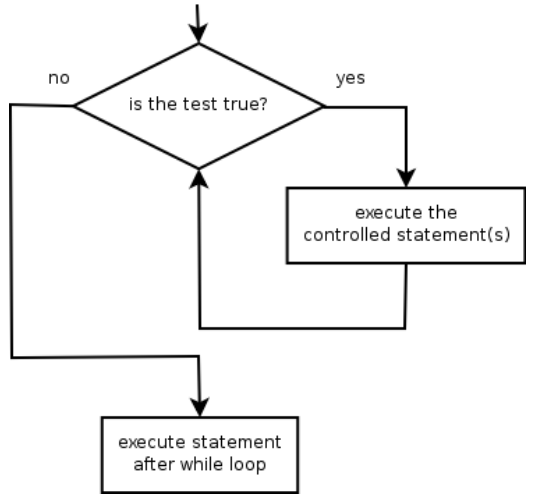
- **definite loop**: Executes a known number of times.
  - The `for` loops we have seen are definite loops.
    - Print "hello" 10 times.
    - Find all the prime numbers up to an integer *n*.
    - Print each odd number between 5 and 127.

- **indefinite loop**: One where the number of times its body repeats is not known in advance.
    - Prompt the user until they type a non-negative number.
    - Print random numbers until a prime number is printed.
    - Repeat until the user has typed "q" to quit.

The `while` loop

- **`while` loop**: Repeatedly executes its body as long as a logical test is true.

```
while (test):
    statement(s)
```



- Example:

```
num = 1
while (num <= 200):
    print(str(num), end=" ")
    num = num * 2

# output:  1 2 4 8 16 32 64 128
```

```
# initialization
# test

# update
```

Example `while` loop

```
# finds the first factor of 91, other than 1
n = 91
factor = 2
while (n % factor != 0):
    factor = factor + 1
print("First factor is " + str(factor))

# output:  First factor is 7
```

- `while` is better than `for` because we don't know how many times we will need to increment to find the factor.

Sentinel values

- **sentinel**: A value that signals the end of user input.
  - **sentinel loop**: Repeats until a sentinel value is seen.

- Example: Write a program that prompts the user for text until the user types "quit", then output the total number of characters typed.
  - (In this case, "quit" is the sentinel value.)

```
Type a word (or "quit" to exit): hello
Type a word (or "quit" to exit): yay
Type a word (or "quit" to exit): quit
You typed a total of 8 characters.
```

Solution?

```
sum = 0
response = "dummy"    # "dummy" value, anything but "quit"

while (response != "quit"):
    response = input('Type a word (or "quit" to exit): ')
    sum = sum  + len(response)

print("You typed a total of " + str(sum) + " characters.")
```

- This solution produces the wrong output. Why?

```
You typed a total of 12 characters.
```

The problem with our code

- Our code uses a pattern like this:

  sum = 0
  while (input is not the sentinel) :
  prompt for input; read input.
  add input length to the sum.

- On the last pass, the sentinel's length (4) is added to the sum:

  prompt for input; read input (`"quit"`*).*
  add input length (4) to the sum.

- This is a fencepost problem.
  - Must read *N* lines, but only sum the lengths of the first *N*-1.

A fencepost solution

```
sum = 0.
prompt for input; read input.              # place a "post"

while (input is not the sentinel):
    add input length to the sum.           # place a "wire"
    prompt for input; read input.          # place a "post"
```

- Sentinel loops often utilize a fencepost "loop-and-a-half" style solution by pulling some code out of the loop.

Correct code

```
sum = 0

# pull one prompt ("fence post") out of the loop
response = input('Type a word (or "quit" to exit): ')

while (response != "quit"):
    sum = sum + len(response)    # moved to top of loop
    response = input('Type a word (or "quit" to exit): ')

print("You typed a total of " + str(sum) + " characters.")
```

Sentinel as a constant

```
SENTINEL = "quit"
...

sum = 0

# pull one prompt ("fence post") out of the loop
response = input('Type a word (or "' + SENTINEL + '" to exit): ')

while (response != SENTINEL):
    sum = sum + len(response)     # moved to top of loop
    response = input('Type a word (or "' + SENTINEL + '" to exit): ')

print("You typed a total of " + str(sum) + " characters.")
```

Strings answer

```
# This program reads a message and a secret key from the user and
# encrypts the message using a Caesar cipher, shifting each letter.
def main():
    message = input("Your secret message: ")
    message = message.lower()
    key = int(input("Your secret key: "))
    encode(message, key)

# This method encodes the given text string using a Caesar
# cipher, shifting each letter by the given number of places.
def encode(text, shift):
    print("The encoded message: ")
    for letter in text:
        # shift only letters (leave other characters alone)
        if (letter >= 'a' and letter <= 'z'):
            letter = chr(ord(letter) + shift)
            # may need to wrap around
            if (letter > 'z'):
                letter = chr(ord(letter) - 26)
            elif (letter < 'a'):
                letter = chr(ord(letter) + 26)
        print(letter, end='')
    print()
```