Introduction to Problem Solving in Python

COSI 10A

Trace `while` loop

- Trace code with x = 1, x = 6, x = 19, x = 39

```
def mystery(x):
    y = 1
    z = 0
    while 2 * y <= x:
        y = y * 2
        z += 1
    print(y, z)
```

Class objectives

- More on Boolean Logic (Section 5.3)
- Fence post problem (Section 5.2)
- Random (second subsection of 3.2)

Boolean logic

Logical operators

- Tests can be combined using logical operators

Operator	Description	Example	Result
`and`	and	`(2 == 3) and (-1 < 5)`	`False`
`or`	or	`(2 == 3) or (-1 < 5)`	`True`
`not`	not	`not (2 == 3)`	`True`

- “Truth tables”

P	q	p and q	p or q
`True`	`True`	`True`	`True`
`True`	`False`	`False`	`True`
`False`	`True`	`False`	`True`
`False`	`False`	`False`	`False`

p	not p
`True`	`False`
`False`	`True`

Using `bool`

- Why type `bool` is useful?
  - Can capture a complex logical test result and use it later
  - Can write a function that does a complex test and returns it
  - Makes code more readable
  - Can pass around the result of a logical test (as param/return)

Returning `bool`

- Write a function that returns True if a given number is prime otherwise False

```
def is_prime(n):
    factors = 0;
    for i in range(1, n + 1):
        if (n % i == 0):
            factors += 1

    if factors == 2:
        return True
    else:
        return False
```

Is this good style?

“Boolean Zen”, part 1

- Students new to `boolean` often test if a result is `True`:

```
if is_prime(57) == True:      # bad
    ...
```

- But this is unnecessary and redundant. Preferred:

```
if is_prime(57):              # good
    ...
```

- A similar pattern can be used for a `False` test:

```
if is_prime(57) == False:     # bad
if not is_prime(57):          # good
```

“Boolean Zen”, part 2

- Functions that return `bool` often have an `if/else` that returns `True` or `False`:

```
def both_odd(n1, n2):
   if n1 % 2 != 0 and n2 % 2 != 0:
        return True
   else:
        return False
```

But the code above is unnecessarily verbose

Solution w/`bool` variable

- We could store the result of the logical test.

```
def both_odd(n1, n2):
    test = (n1 % 2 != 0 and n2 % 2 != 0)
    if test:    # test == True
        return True
    else:       # test == False
        return False
```

- Notice: Whatever `test` is, we want to return that
  - If `test` is `True`, we want to return `True`
  - If `test` is `False`, we want to return `False`

Solution w/`bool` variable

- Observation: The `if/else` is unnecessary.
  - The variable `test` stores a `bool` value; its value is exactly what you want to return. So return that!

```
def both_odd(n1, n2):
     test = (n1 % 2 != 0 and n2 % 2 != 0)
     return test
```

- An even shorter version:
  - We don't even need the variable `test`. We can just perform the test and return its result in one step

```
def both_odd(n1, n2):
       return (n1 % 2 != 0 and n2 % 2 != 0)
```

“Boolean Zen” template

- Replace

```
def name(parameters):
    if test:
        return True
    else:
        return False
```

- With

```
def name(parameters):
    return test
```

Returning `bool`

- Write a function that returns True if a given number is prime otherwise False

```
def is_prime(n):
    factors = 0;
    for i in range(1, n + 1):
        if (n % i == 0):
            factors += 1

    if factors == 2:
        return True
    else:
        return False
```

Is this good style?

```
def is_prime(n):
    factors = 0;
    for i in range(1, n + 1):
        if (n % i == 0):
            factors += 1

    return factors == 2
```

Fence post problem

Problem

- Write a function `print_letters` that prints each letter from a word separated by commas
- For example, the call to `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(word[i] + ", ", end='')
      print()   # end line
```

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

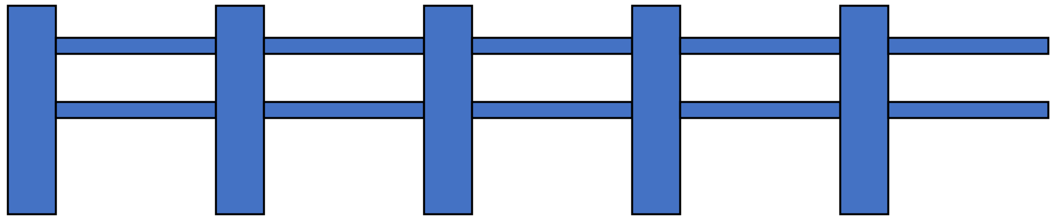
```
def print_letters(word):
      for i in range(0, len(word)):
         print(", " + word[i], end='')
      print()   # end line
```

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 solution

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

- Alternate solution:

```
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
```

Sentinel values

- Write a program that prompts the user for text until the user types “quit”, the output the total number of characters types

```
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.
```

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

Sentinel values solution1

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

while response != "quit":
    response = input("Type a word (or \"quit\" to exit): ")
    sum += len(resspone)
print("You typed a total of " + str(sum) + " characters.")
```

- Does this work? Why?

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 style solution by pulling some code out of the loop

Sentinel values solution1 (correct)

```python
sum = 0

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

while (response != "quit"):
    sum += len(response)
    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/read ("post") out of the loop
response = input("Type a word (or \"" + SENTINEL + "\" to exit): ")

while response != SENTINEL:
    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.")
```

Random

`random` module

- Python’s standard libraries include the random module that contains useful functions to generate random numbers
- Technically the numbers generated by the library are called **pseudo-random** because they are actually based on mathematical functions and system clock
- To use the random library module, you need the import statement

```
import random
```

Python's random module

Method name	**Description**
`random.random()`	returns a random float in the range [0, *1*) in other words, 0 inclusive to *1* exclusive
`random.randint(`***min, max***`)`	returns a random integer in the range [min, *max*] in other words, min to *max* inclusive

`import random` is necessary to use the **above** functions

Example:

```
import random
random_number = random.randint(1, 10)    # 1-10
random_number = random.randint(4, 10)    # 4-10
```

Programming Question1

- Write a function that repeatedly flips a coin until the result of the coin toss are three heads.

```
T T H T T T H T H T H H H
Three heads in a row!
```