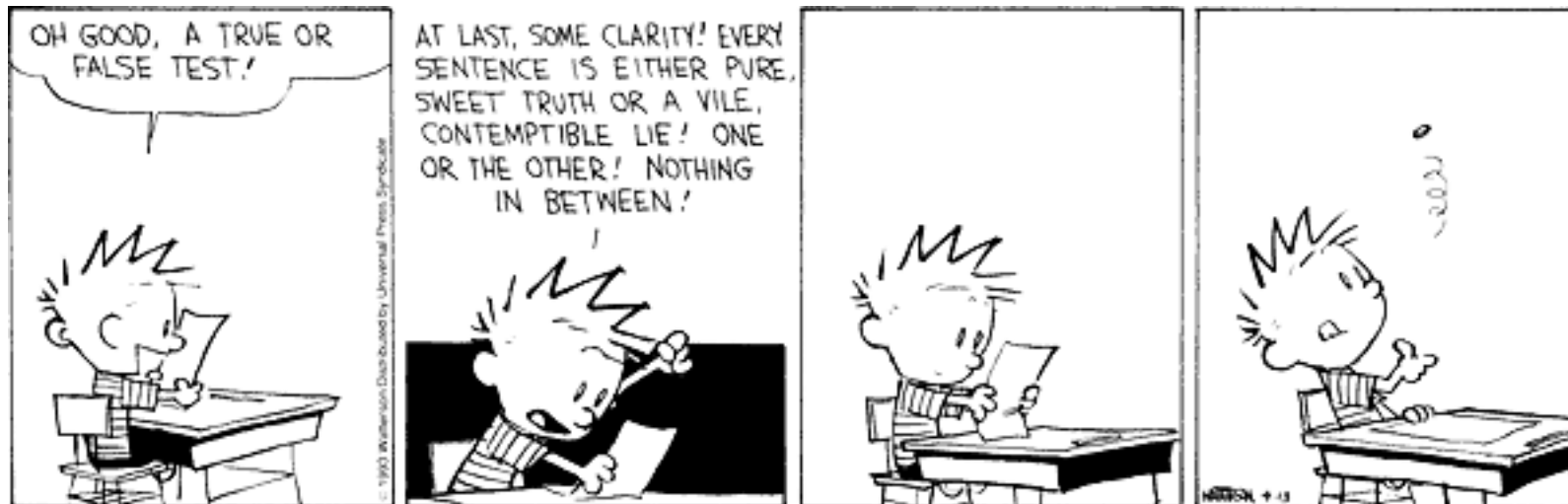


CSc 110, Spring 2018

Lecture 13: Cumulative Sum and Boolean Logic

Adapted from slides by Marty Stepp and Stuart Reges



Adding many numbers

- How would you find the sum of all integers from 1-1000?

```
# This may require a lot of typing
```

```
sum = 1 + 2 + 3 + 4 + ...
```

```
print("The sum is", sum)
```

- What if we want the sum from 1 - 1,000,000?
Or the sum up to any maximum?
 - How can we generalize the above code?

Cumulative sum loop

```
sum = 0
for i in range(1, 1001):
    sum = sum + i

print("The sum is", sum)
```

- **cumulative sum:** A variable that keeps a sum in progress and is updated repeatedly until summing is finished.
 - The `sum` in the above code is an attempt at a cumulative sum.
 - Cumulative sum variables must be declared *outside* the loops that update them, so that they will still exist after the loop.

Cumulative product

- This cumulative idea can be used with other operators:

```
product = 1
for i in range(1, 21):
    product = product * 2

print("2 ^ 20 =", product)
```

- How would we make the base and exponent adjustable?

input and cumulative sum

- We can do a cumulative sum of user input:

```
sum = 0
for i in range(1, 101):
    next = int(input("Type a number: "))
    sum = sum + next

print("The sum is", sum)
```

Cumulative sum question

- Modify the `receipt` program from lecture 2
 - Prompt for how many people, and each person's dinner cost.
 - Use functions to structure the solution.
- Example log of execution:

```
How many people ate? 4
Person #1: How much did your dinner cost? 20.00
Person #2: How much did your dinner cost? 15
Person #3: How much did your dinner cost? 30.0
Person #4: How much did your dinner cost? 10.00
```

```
Subtotal: $75.0
Tax: $6.0
Tip: $11.25
Total: $92.25
```

Cumulative sum answer

```
# This program enhances our Receipt program using a cumulative sum.
```

```
def main():
```

```
    subtotal = meals()
```

```
    results(subtotal)
```

```
# Prompts for number of people and returns total meal subtotal.
```

```
def meals():
```

```
    people = float(input("How many people ate? "))
```

```
    subtotal = 0.0;                # cumulative sum
```

```
    for i in range(1, people + 1):
```

```
        person_cost = float(input("Person #" + str(i) +  
                                   ": How much did your dinner cost? "))
```

```
        subtotal = subtotal + person_cost # add to sum
```

```
    return subtotal
```

```
...
```

Cumulative answer, cont'd.

```
# Calculates total owed, assuming 8% tax and 15% tip
```

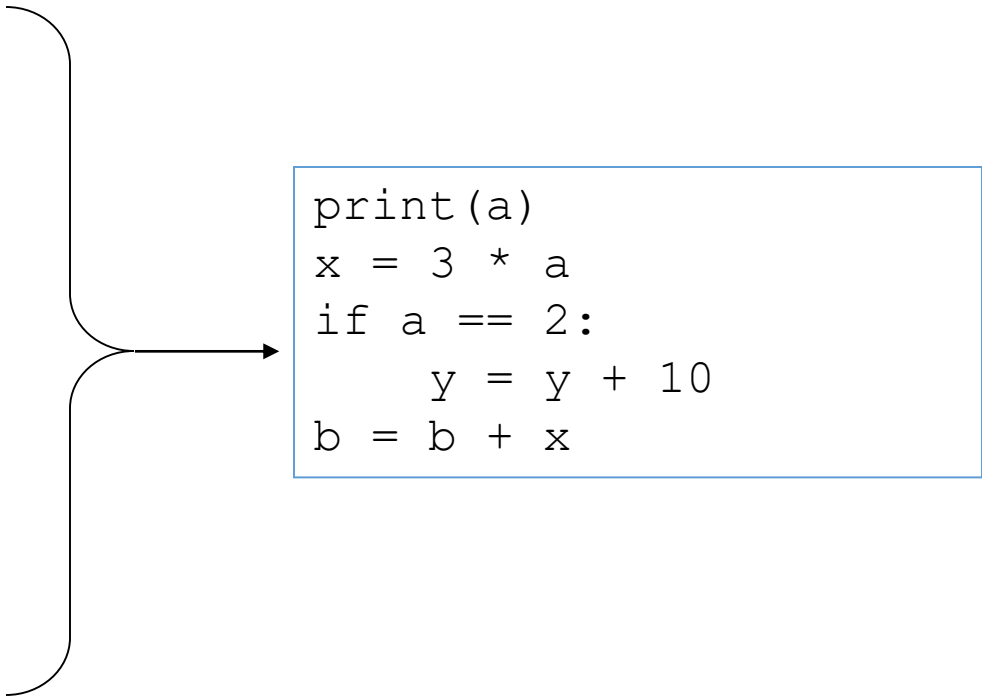
```
def results(subtotal):  
    tax = subtotal * .08  
    tip = subtotal * .15  
    total = subtotal + tax + tip  
  
    print("Subtotal: $" + str(subtotal))  
    print("Tax: $" + str(tax))  
    print("Tip: $" + str(tip))  
    print("Total: $" + str(total))
```


Factoring `if/else` code

- **factoring:** Extracting common/redundant code.
 - Can reduce or eliminate redundancy from `if/else` code.

- **Example:**

```
if a == 1:
    print(a)
    x = 3
    b = b + x
elif a == 2:
    print(a)
    x = 6
    y = y + 10
    b = b + x
else:           # a == 3
    print(a)
    x = 9
    b = b + x
```



```
print(a)
x = 3 * a
if a == 2:
    y = y + 10
b = b + x
```

Relational expressions

- `if` statements use logical tests.

```
if i <= 10: ...
```

- These are Boolean expressions.
- Tests use *relational operators*:

Operator	Meaning	Example	Value
<code>==</code>	equals	<code>1 + 1 == 2</code>	True
<code>!=</code>	does not equal	<code>3.2 != 2.5</code>	True
<code><</code>	less than	<code>10 < 5</code>	False
<code>></code>	greater than	<code>10 > 5</code>	True
<code><=</code>	less than or equal to	<code>126 <= 100</code>	False
<code>>=</code>	greater than or equal to	<code>5.0 >= 5.0</code>	True

Logical operators

- Tests can be combined using *logical operators*:

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

- "Truth tables" for each, used with logical values p and q :

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

Evaluating logical expressions

- Relational operators have lower precedence than math; logical operators have lower precedence than relational operators

5 * 7 >= 3 + 5 * (7 - 1) and 7 <= 11

5 * 7 >= 3 + 5 * 6 and 7 <= 11

35 >= **3 + 30** and 7 <= 11

35 >= 33 and 7 <= 11

True and True

True

Logical questions

- What is the result of each of the following expressions?

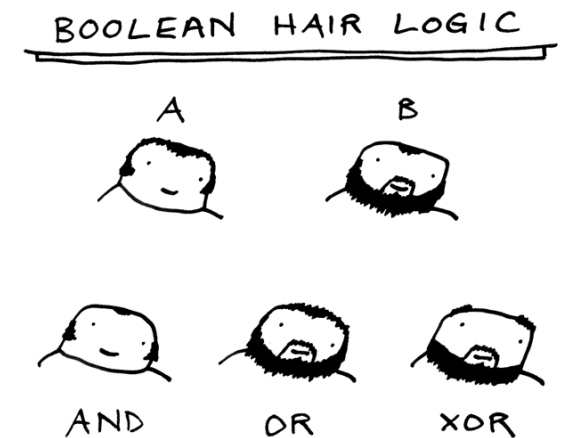
$x = 42$

$y = 17$

$z = 25$

- $y < x$ and $y \leq z$
- $x \% 2 == y \% 2$ or $x \% 2 == z \% 2$
- $x \leq y + z$ and $x \geq y + z$
- $\text{not}(x < y \text{ and } x < z)$
- $(x + y) \% 2 == 0$ or $\text{not}((z - y) \% 2 == 0)$

- Answers: True, False, True, True, False



Type `bool`

- **`bool`**: A logical type whose values are `True` and `False`.
 - A logical *test* is actually a Boolean expression.
 - Like other types, it is legal to:
 - create a `bool` variable
 - pass a `bool` value as a parameter
 - return a `bool` value from function
 - call a function that returns a `bool` and use it as a test

```
minor      = age < 21
is_prof    = "Prof" in name
loves_csc  = True
```

```
# allow only CS-loving students over 21
if minor or is_prof or not loves_csc:
    print("Can't enter the club!")
```

Returning bool

```
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?



- Calls to functions returning `bool` can be used as tests:

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

"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/ "Boolean Zen"

- 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
```

Improve the `is_prime` function

- How can we fix this code?

```
def is_prime(n):  
    factors = 0;  
    for i in range(1, n + 1):  
        if n % i == 0:  
            factors += 1  
  
    if factors != 2:  
        return False  
    else:  
        return True
```

De Morgan's Law

- **De Morgan's Law:** Rules used to negate boolean tests.
 - Useful when you want the opposite of an existing test.

Original Expression	Negated Expression	Alternative
<code>a and b</code>	<code>not a or not b</code>	<code>not(a and b)</code>
<code>a or b</code>	<code>not a and not b</code>	<code>not(a or b)</code>

- Example:

Original Code	Negated Code
<pre>if x == 7 and y > 3: ...</pre>	<pre>if x != 7 or y <= 3: ...</pre>

Boolean practice questions

- Write a function named `is_vowel` that returns whether a `str` is a vowel (a, e, i, o, or u), case-insensitively.
 - `is_vowel("q")` returns `False`
 - `is_vowel("A")` returns `True`
 - `is_vowel("e")` returns `True`
- Change the above function into an `is_non_vowel` that returns whether a `str` is any character except a vowel.
 - `is_non_vowel("q")` returns `True`
 - `is_non_vowel("A")` returns `False`
 - `is_non_vowel("e")` returns `False`

Boolean practice answers

```
# Enlightened version. I have seen the true way (and false way)
```

```
def is_vowel(s):
```

```
    return s == 'a' or s == 'A' or s == 'e' or s == 'E' or s == 'i' or s == 'I'  
           or s == 'o' or s == 'O' or s == 'u' or s == 'U'
```

```
# Enlightened "Boolean Zen" version
```

```
def is_non_vowel(s):
```

```
    return not(s == 'a') and not(s == 'A') and not(s == 'e') and not(s == 'E')  
           and not(s == 'i') and not(s == 'I') and not(s == 'o') and  
           not(s == 'O') and not(s == 'u') and not(s == 'U')
```

```
# or, return not is_vowel(s)
```


When to return?

- Functions with loops and return values can be tricky.
 - When and where should the function return its result?
- Write a function `seven` that uses `randint` to draw up to ten lotto numbers from 1-30.
 - If any of the numbers is a lucky 7, the function should stop and return `True`. If none of the ten are 7 it should return `False`.
 - The method should print each number as it is drawn.

```
15 29 18 29 11 3 30 17 19 22 (first call)
29 5 29 4 7 (second call)
```

Flawed solution

```
# Draws 10 lotto numbers; returns True if one is 7.
def seven():
    for i in range(10):
        num = randint(1, 30)
        print(num, " ", end='')

        if num == 7:
            return True
        else:
            return False
```

- The function always returns immediately after the first draw.
- This is wrong if that draw isn't a 7; we need to keep drawing.

Returning at the right time

```
# Draws 10 lotto numbers; returns True if one is 7.
def seven():
    for i in range(1, 11):
        num = randint(1, 30)
        print(str(num) + " ", end='')

        if num == 7:      # found lucky 7; can exit now
            return True

    return False      # if we get here, there was no 7
```

- Returns `True` immediately if 7 is found.
- If 7 isn't found, the loop continues drawing lotto numbers.
- If all ten aren't 7, the loop ends and we return `False`.

if/else, return question

- Write a function `count_factors` that returns the number of factors of an integer.
 - `count_factors(24)` returns 8 because 1, 2, 3, 4, 6, 8, 12, and 24 are factors of 24.

- Solution:

```
# 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 += 1          # i is a factor of number  
    return count
```