Welcome!

We'll get started shortly ...

CS 49 Section

Week 6

Surajit A Bose

Agenda

- Logistics and check-ins
- Review of lecture concepts
  - Boolean expressions
  - Control flow
- Section Problems:
  - Dog Year Calculator
  - Finding Factors
  - Rock, Paper, Scissors

Logistics

How to get hold of me / get help+

- The class forum or section forum, 24 hr turnaround
  - Feel free not only to ask, but also to answer questions there!
- Surajit's office hours:
  - Tuesdays 1p–2p on Zoom
  - By appointment on Zoom or on campus
- Lane's office hours
- Canvas inbox for Lane or Surajit
- Email bosesurajit@fhda.edu, 24 hr turnaround
- Online or in-person tutoring via the STEM center (Room 4213)
- The section GitHub repo has lecture and section slides and solutions

Check In

Any questions about:

- The **`random`** module and/or the **`math`** library
- Variables (names, types, values, assignment, casting)
- Constants
- Arithmetic operators and precedence (**`**`**, unary **`-`**, **`*`**, **`/`**, **`//`**, **`%`**, **`+`**, **`-`**)
- Compound operators (**`+=`**, **`-=`**, **`*=`**, etc)
- Problems from the homework or extra credit
- Anything else?

Please take the Zoom poll! And thanks to those who filled out the survey!

Lecture Review: Booleans

Boolean Expressions

- Reminder: an expression is a statement that can be evaluated
- A boolean expression evaluates to one of two values: `True` or `False`
- Like arithmetic expressions:
  - Boolean expressions are built using operators
  - The result is typically stored in a variable
  - For example, given a whole number greater than 1, we could set a variable **`is_prime`** to the value `True` or `False` for that number
- Boolean operators are of two types:
  - Comparison
  - Logical

Boolean Operators: Comparison

- The boolean comparators are similar to the ones in math
- Given `x = 3` and `y = 4`:
  - `x == y - 1` *# x equals y minus 1*
  - `x + 1 != y` *# x plus 1 not equal to y*
  - `x < y` *# x less than y*
  - `x <= y` *# x less than or equal to y*
  - `x > y` *# x greater than y*
  - `x >= y` *# x greater than or equal to y*

Boolean Operators: Comparison

- The boolean comparators are similar to the ones in math
- Given `x = 3` and `y = 4`:
  - `x == y - 1` `# x equals y minus 1` `True`
  - `x + 1 != y` `# x plus 1 not equal to y` `False`
  - `x < y` `# x less than y` `True`
  - `x <= y` `# x less than or equal to y` `True`
  - `x > y` `# x greater than y` `False`
  - `x >= y` `# x greater than or equal to y` `False`

Boolean Operators: Comparison

- Note that `==` and `=` are very different beasts!
  - `x == y - 1` is not the same as `x = y - 1`
  - One is an expression: it checks whether the operands on the left and right sides of the operator are equal
  - The other is an assignment: it evaluates the expression on the RHS and stores the resulting value into the variable on the LHS
- Comparison operators can be chained:
  - `x < y` and `y < z` can be expressed as `x < y < z`, to check whether the value of **y** is between the values of **x** and **z**

Any Questions?

Boolean Operators: Logical

- The logical operators are **not**, **and**, and **or**
- **not** simply reverses the truth value of its operand:
  - `3 > 4 is False`, so `not (3 > 4)` is `True`
  - if `x < y` is `True`, then `not (x < y)` is `False`
  - if `x < y` is `True`, then `not (x >= y)` is `True`
- Note the opposites:
  - The opposite of `==` is `!=`
  - The opposite of `<` is `>=`
  - The opposite of `>` is `<=`
- So if `x < 5` is `False`, then `not (x < 5)` is `True` and `x >= 5` is `True`

Boolean Operators: Logical

- The logical operators are `not`, `and`, and `or`
- `and` is `True` only when both of its operands are `True`:
  - `3 < 5` and `5 >= 4` are both `True`, so `(3 < 5) and (5 >= 4)` is `True`
  - `3 < 5` is `True` but `5 < 4` is `False`, so `(3 < 5) and (5 < 4)` is `False`
- `or` is `True` when at least one of the operands is True:
  - `3 < 5` and `5 >= 4` are both `True`, so `(3 < 5) or (5 >= 4)` is `True`
  - `3 < 5` is `True` and `5 < 4` is `False`, but `(3 < 5) or (5 < 4)` is `True`
  - `3 > 5` is `False` and `5 < 4` is `False`, so `(3 > 5) or (5 < 4)` is `False`
- Short circuiting evaluation: operands are evaluated from left to right, so
  - for `and`, if the left operand is `False`, the right operand is never evaluated
  - for `or`, if the left operand is `True`, the right operand is never evaluated

Boolean Operators: Logical

- Given short circuiting, what will this code print to the screen?

```
def main():
    print(True or 'Hello')
    print(False or 'Hello')
    print(True and 'Hello')
    print(False and 'Hello')

if __name__ == '__main__':
    main()
```

Boolean Operators: Logical

- Given short circuiting, what will this code print to the screen?

```
def main():
    print(True or 'Hello')       # True
    print(False or 'Hello')      # Hello
    print(True and 'Hello')      # Hello
    print(False and 'Hello')     # False

if __name__ == '__main__':
    main()
```

Any Questions?

Operator Precedence

From highest to lowest:

- Parentheses
- Arithmetic operators:
  - Exponentiation
  - Unary negation
  - Multiplication, division, integer division, modulus
  - Addition, subtraction
- Comparison operators: equals, less than, etc, all identical in precedence
- Logical not
- Logical and, or

Operator Precedence

- Use parentheses rather than relying on implicit precedence
  - `x + (y / z)` is clearer than `x + y / z`, even though they are equivalent
- Watch out for the higher precedence of logical `not`!
- If `x` and `y` are both `False`, what is the value of:
  - `not x and y`
  - `not (x and y)`
  - `not x or y`
  - `not (x or y)`

Operator Precedence

- Use parentheses rather than relying on implicit precedence
  - `x + (y / z)` is clearer than `x + y / z`, even though they are equivalent
- Watch out for the higher precedence of logical `not`!
- If `x` and `y` are both `False`, what is the value of:
  - `not x and y` `# (not False) and False` `False`
  - `not (x and y)` `# not (False and False)` `True`
  - `not x or y` `# (not False) or False` `True`
  - `not (x or y)` `# not (False or False)` `True`

Any Questions?

Lecture Review: Control Flow

Flashback to Week 2 ...

Control Flow Overview

- `while` loop: Continuously performs a block of code until a given boolean expression (aka the loop condition) is evaluated to `False`
- `if` statement: Performs a block of code only when a boolean expression (aka the `if` condition) is `True`, and only once
- `if-else` statement: Performs a block of code when a boolean expression is `True`, or a different block (aka the `else` condition) when the expression is `False`. Either block is performed only once
- `if-elif-else` statement: Performs one of three or more blocks of code depending on which boolean expression is `True`. Only one block of code is performed, and only once
- `for` loop: Performs some block of code a specific number of times

`while` Loop

- The `while` loop is governed by a boolean expression
- If the expression evaluates to `False`, the loop is never entered and any code inside the loop is ignored
- If the expression evaluates to `True`, the loop is entered and the block of code inside it is executed
- At the end of the code block, the boolean is re-evaluated. If the expression is still `True`, then the loop is re-entered.
- This continues until the boolean evaluates to `False`.
- This loop is also called an *indefinite loop* because it will run until the associated condition becomes `False`.

`while` Loop

- Beware of infinite loops, where the boolean will never be False!
- Something inside the loop body must eventually alter the loop condition.
- The following code fragment is intended to allow employee lookups based on employee ID numbers. The user enters employee ID numbers one after another, entering '-1' when all lookups are done:

```
employee_id = input("Enter employee ID number: ")
while int(employee_id) != -1 :
    # Some code here to process employee in some way
    next_id = input("Enter employee ID number: ")
```

while Loop

- Beware of infinite loops, where the boolean will never be False!
- Something inside the loop body must eventually alter the loop condition.
- The following code fragment is intended to allow employee lookups based on employee ID numbers. The user enters employee ID numbers one after another, entering '-1' when all lookups are done:

```
employee_id = input("Enter employee ID number: ")
while int(employee_id) != -1 :
    # Some code here to process employee in some way
    employee_id = input("Enter employee ID number: ")
```

`while` Loop

- In this example, the loop continues until the user enters -1.
- This value is called a sentinel, as it guards against the loop going on forever by signalling that the loop should terminate
- Sentinels are often declared as constants

```
SENTINEL = -1
def main():
    employee_id = input("Enter employee ID number: ")
    while int(employee_id) != SENTINEL :
        # Some code here to process employee in some way
        employee_id = input("Enter employee ID number: ")
```

Any Questions?

`if-elif-else` Statement

- The `if-elif-else` construction allows us to check multiple conditions
- For example, to determine whether a given year is a leap year:

```
if year % 4 :
    is_leap = False
elif (year % 100 == 0) and (year % 400) :
    is_leap = False
else :
    is_leap = True
```

- Note: `if year % 4` is equivalent to `if (year % 4) != 0`

`if-elif-else` Statement

- What is wrong with this code?

```
x = 5
y = 12
if x = y:
    print("They're equal")
elif x < y:
    print("x is smaller!")
else:
    print("y is smaller!")
```

`if-elif-else` Statement

- What is wrong with this code?

```
x = 5
y = 12
if x == y:
    print("They're equal")
elif x < y:
    print("x is smaller!")
else:
    print("y is smaller!")
```

Any Questions?

`for` Loop

- The `for` loop performs its block of code a specified number of times
- It is controlled by a loop variable, usually `i` (for index)
- `i` is often specified as a `range`
- Syntax: `for i in range(start, stop, step)`
  - Default start is 0
  - A specified stop is always required
  - Default step is 1; if a different one is needed, must specify start too
- `i` enters the `for` loop with value 0 or the specified start, increments by 1 or the specified step, and terminates the loop when `i == stop`
- Note that the loop does not execute when `i` has the value of `stop`.

for Loop

- Example: First 10 odd numbers

```
for i in range(1, 21, 2):
    print(i)
```

- Example: Countdown

```
for i in range(10, 0, -1):
    print(i)
print("Blast off!")
```

Any Questions?

Section problem:
Dog Year Calculator

https://codeinplace.stanford.edu/foothill-cs49/ide/a/dogyearssection

Dog Year Calculator

- One dog year is the equivalent of seven human years. Write a Python program that prompts the user to enter an age in human years. Output the equivalent age in dog years:

```
Enter an age in human years: 12
The age in dog years is 84
```

- Check that the user enters a positive integer
- Continue prompting for a human age until the user enters zero
- What constants should we use?
- As what type should the input from the user be cast?

Dog Year Calculator

Here is an example of what one run of your program should look like (user input is *italicized*):

> Enter an age in human years: *-12*
> Sorry, please enter a positive number or 0 to exit
> Enter an age in human years: *13*
> The age in dog years is 91
> Enter an age in human years: *0*

Section problem: Finding Factors

https://codeinplace.stanford.edu/foothill-cs49/ide/a/findingfactors

Finding Factors

- Ask the user to enter an integer
- Check that the number is greater than zero
- Print all the factors of the given number one by one
- Ask for another number and repeat until the user enters zero
- What constant should we use?
- How do we obtain the factors?

Here is an example of what one run of your program should look like (user input is *italicized*):

> Your number: *-10*
> Please input a positive number
> Your number: *42*
> 1
> 2
> 3
> 6
> 7
> 14
> 21
> 42
> Your number: *53*
> 1
> 53
> Your number: *0*

That's all, folks!

Next up: Graphics!