Welcome!

We'll get started shortly. Please take the Zoom poll in the meanwhile!

CS 49 Week 5

Surajit A. Bose

Agenda

- Logistics
- Boolean expressions
  - Comparison operators
  - Logical operators
- Conditionals: **`if`**, **`if-else`**, **`if-elif-else`**
  - Worked example: Check for leap year
- **`while`** loops
- **`for`** loops
  - Worked example: First 20 even numbers
- Section problem: High-Low

Logistics

Reminders

- General expectation during section is that cameras stay on
- Complete Mid-Course Survey this week
- Check Pronto for section communications
    - Launch Pronto from the left hand navigation bar of the course Canvas page
    - Open Pronto into a full browser window
    - If the list of groups for the course is not showing, click on the course title
    - Click on the "CS49 Section – Surajit" group
    - Typically, three announcements weekly
        - Toward beginning of week: what's coming up in section
        - A few minutes before section: a reminder to join
        - Shortly after section: recap
    - DM me via Pronto for help with course material or assignments

How to get hold of me / get help from other resources

- Surajit's office hours
  - Fridays 12 noon–1p, directly after section
  - By appointment on Zoom
- Pronto DM for a quick response (usually within a couple hours)
- Email bosesurajit@fhda.edu or Canvas inbox, 24 hr turnaround
- The github repo has section materials, starter code, and solutions

---

- Pronto DM or Canvas inbox for Lane
- Lane's office hours
- Online or in-person tutoring at the STEM center (Room 4213)

Boolean expressions

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

Comparison operators (Slide 1 of 3)

- The boolean comparators are similar to the ones in math
- Given **`x = 3`** and **`y = 4`**, what is the value of these boolean expressions?
  - `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`

Comparison operators (Slide 2 of 3)

- The boolean comparators are similar to the ones in math
- Given **`x = 3`** and **`y = 4`**, what is the value of these boolean expressions?
  - **`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**

Comparison operators (Slide 3 of 3)

- 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`**

Logical operators (Slide 1 of 2)

- 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`

Logical operators (Slide 2 of 2)

- 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

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`**
- Logical **`or`**

Conditionals

Conditionals: `if` and `if-else`

- `if` statement:
    - When the associated boolean expression is `True`, the indented block below the `if` clause runs once
    - When the boolean is `False`, the indented block does not run
- `if-else` statement
    - When the associated boolean expression is `True`, the indented block below the `if` clause runs once
    - When the boolean is `False`, the indented block below the `else` clause runs once

Conditionals: `if-elif-else`

- `if-elif-else` is used when multiple conditions need to be checked
- Each condition is checked one after another until:
  - One of them evaluates to `True`, in which case only the block of code indented below that `if` or `elif` clause is executed, and only once
  - All of them have evaluated to `False`, in which case only the block of code below the `else` clause is executed, and only once
- The `else` clause is not required: could have just `if` and `elif` clauses
- Example: checking for leap years
  - If a given year is not evenly divisible by 4, it is not a leap year: 2024 was a leap year, 2025 is not
  - If a given year is evenly divisible by 100, it is not a leap year unless it is also divisible by 400: 2000 was a leap year, 1900 was not

A conditional puzzle (Slide 1 of 2)

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!")
```

A conditional puzzle (Slide 2 of 2)

The correct code:

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

Loops

The `while` loop (Slide 1 of 2)

- 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`.

The `while` loop (Slide 2 of 2)

- `while` loops can be used to check user input, for example to control game play:

```
keep_going = True
while keep_going:
    # All the game commands
    # At the end of the round:
    wants_more = input("Play another round? y/n: ")
    if wants_more == "n":
        keep_going = False
```

- Here, `"n"` is called a **sentinel value** as it guards against the loop running forever.

The `for` loop (Slide 1 of 2)

- The **`for`** loop performs its block of code a specified number of times
- It is called a *definite loop* as the number of times it runs is predictable
- It is controlled by a loop variable, often **`i`** (for index)
- **`i`** is typically 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 **`step`** is needed, **`start`** must also be specified
- **`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`**

The for loop (Slide 2 of 2)

- Example: First 20 even numbers

```
for i in range(20):
    print(i * 2)
```

- Example: First 10 odd numbers

```
for i in range(1, 20, 2):
    print(i)
```

- Example: Lost marbles

```
for i in range(10, 1, -1):
    print(f"I have {i} marbles left. Oops, there goes another!")
print("I'm down to my last marble!")
```

Section Problem: High-Low

High-Low: Overview

- Two numbers are generated from 1 to 100 (inclusive on both ends): one for you and one for the computer, who will be your opponent
- You get to see your number, but not the computer's
- You make a guess, saying your number is either higher than or lower than the computer's number
- If your guess matches the truth (e.g. you guess your number is higher, and then your number is actually higher than the computer's), you get a point
- Design the game following the milestones in the assignment description!
- Note: if you do Extension 2, the autograder will mark your code wrong. This is a known issue.

That's all, folks!

Next up: Graphics

Bonus Slides

More about `while` loops, booleans, precedence

More about `while` loops

Infinite loops (Slide 1 of 2)

- 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` as a sentinel when all lookups are done:

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

Infinite loops (Slide 2 of 2)

- 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` as a sentinel when all lookups are done:

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

More about `True` and `False`

Truthy and falsy values

- Some Python values are automatically considered **`False`**, e.g.
  - Zero, whether as **`int 0`** or **`float 0.0`**
  - Empty strings **`""`**
  - Other empty sequences like empty lists or dictionaries (we'll cover lists and dictionaries in the last week)
  - **`False`** itself
- Conversely, their opposites are considered **`True`**, e.g.
  - Any nonzero number
  - Any non-empty string or sequence
  - **`True`** itself

Using zero and nonzero

- Instead of

  ```
  if year % 4 != 0:
  ```

  can write

  ```
  if year % 4:
  ```

- Instead of

  ```
  if year % 100 == 0:
  ```

  can write

  ```
  if not year % 100:
  ```

Short-circuiting and strings (Slide 1 of 2)

- Given short-circuit evaluation of **and** and **or** and the truthiness of non-empty strings, what will the following code print?

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

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

Short-circuiting and strings (Slide 2 of 2)

- Given short-circuit evaluation of **and** and **or** and the truthiness of non-empty strings, what will the following code print?

```
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()
```

More about operator precedence

Operator Precedence (Slide 1 of 2)

- Use parentheses rather than relying on implicit precedence
  - Even though they are equivalent, **`x + (y / z)`** is clearer than **`x + y / z`**
- 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 (Slide 2 of 2)

- Use parentheses rather than relying on implicit precedence
  - Even though they are equivalent, `x + (y / z)` is clearer than `x + y / z`
- 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**

Bonus Slides

Advanced Topics

Advanced topic: `match case`

match case

- A possible alternative to **if-elif-else** is **match case**
- Note: **match case** does not work in the Code in Place IDE
- Example: song If it's Tuesday, this must be Belgium

```
if day == "Tuesday":
    place = "Belgium"
elif day == "Wednesday":
    place = "Rome"
elif day == "Thursday":
    place = "Montreux"

print (f"If it's {day} it must be {place}")
```

match case

- A possible alternative to **if-elif-else** is **match case**
- Python tries to match the value to a specific case to see what commands to run
- Example: song If it's Tuesday, this must be Belgium

```
if day == "Tuesday":
    place = "Belgium"
elif day == "Wednesday":
    place = "Rome"
elif day == "Thursday":
    place = "Montreux"
```

```
match day:
    case "Tuesday":
        place = "Belgium"
    case "Wednesday":
        place = "Rome"
    case "Thursday":
        place = "Montreux"
```

```
print (f"If it's {day} it must be {place}")
```

Advanced topic: bitwise operators

Bitwise operators (Slide 1 of 4)

- In addition to the logical operators **`and`** and **`or`**, Python also has three bitwise operands:
  - bitwise not **`~`**
  - bitwise and **`&`**
  - bitwise or **`|`**
- These operate on the binary representations of the operand values
- A binary representation is a representation in base 2
- An explanation of base 2 representation is in this Khan Academy video
- The bitwise operators work in ways analogous to when the logical **`not`**, **`and`**, and **`or`** return **`True`** or **`False`**

Bitwise operators (Slide 2 of 4)

- bitwise `~` evaluates to `0` if the operand bit is `1`, `1` if the operand bit is `0`

```
~1      # result: 0                ~0      # result: 1
```

- bitwise `&` evaluates to `1` if both operand bits are `1`, and `0` if either operand bit is `0`

```
1 & 1   # result: 1                0 & 1   # result: 0
1 & 0   # result: 0                0 & 0   # result: 0
```

- bitwise `|` evaluates to `0` if both operand bits are `0`, and `1` if either operand bit is `1`

```
1 | 1   # result: 1                1 | 0   # result: 1
0 | 1   # result: 1                0 | 0   # result: 0
```

Bitwise operators (Slide 3 of 4)

```
def bitwise_operations():
    a = 3        # decimal 3: binary    011
    b = 6        # decimal 6: binary    110
    c = a & b    # result: ?
    d = a | b    # result: ?

    print(f"Result of bitwise and, 3 & 6 : {c}")
    print(f"Result of bitwise or,  3 | 6 : {d}")

if __name__ == "__main__":
    bitwise_operations()
```

Bitwise operators (Slide 4 of 4)

```
def bitwise_operations():
    a = 3        # decimal 3: binary    011
    b = 6        # decimal 6: binary    110
    c = a & b    # result:          &   010    decimal 2
    d = a | b    # result:          |   111    decimal 7

    print(f"Result of bitwise and, 3 & 6 : {c}")
    print(f"Result of bitwise or,  3 | 6 : {d}")

if __name__ == "__main__":
    bitwise_operations()
```