L05: Conditionals

Eric Franzosa, Ph.D.

franzosa@hsph.harvard.edu

Outline

- HW2 reminder
- Boolean values
- Comparative and logical operators
- Conditionals: `if/elif/else`
- The `while` loop and loop control
- Practice

`True` and `False`

- There are only two Boolean values: `True` and `False`
  - Used by computers to track objectively true and false statements
  - Sometimes represented as 1 and 0

- "The blue whale is the largest mammal"
  - **Objectively true**

- "Texas is the largest of the 50 United States"
  - **Objectively false**

- "The black bear is the best type of bear"
  - **Opinion - Neither objectively true nor false.**

Comparative operators

Comparative operators return `True`/`False` values:

- == : test of equality
- `!=` : test of inequality ("not equals")
- < and > : less-than / greater-than
- <= and >= : less-than-or-equal-to / greater-than-or-equal-to

In [ ]:
```
# == returns True if the surrounding values are equal...
5 == 5
```

In [ ]:
```
# ...and False otherwise
5 == -5
```

In [ ]:
```
# != returns the opposite of ==
5 != -5
```

Be careful not to confuse = and ==.

- = is the assignment operator: puts data into a variable.
- == is the test of equality: evaluates if two pieces of data are equal.
- Very common programming mix-up.

Note that neither = nor == are direct analogs of **=** as it's used in math.

- **=** in math is an assertion: x = 5 implies x **is** 5.
- == poses a question: `x == 5` asks "Is x equal to 5?" (it might not be).
- = is an action: `x = 5` sets the value of x to 5.

Aside: Checking divisibility with == with %

- % is the modulus operator.
- returns the remainder when we divide x by y.
- x % y is read "x mod y" (similar to how x * y is read "x times y").

In [ ]:
```
5 % 2
```

- If **x mod y** is 0, then **x** divides evenly into **y**.
- For example, if **x mod 2** is 0, then **x** is even.
- We will use this *a lot* today.

In [ ]:
```
10 % 2 == 0
```

Comparative operators: <, >, <=, >=

In [ ]:
```
2 < 3
```

In [ ]:
```
3 < 3
```

In [ ]:
```
3 <= 3
```

Comparative operators work on strings as well (where they indicate lexical order). It can be helpful to think of `x < y` as meaning "does `x` come before `y` in sorted list?"

In [ ]:
```
"a" < "b"
```

In [ ]:
```
"b" < "a"
```

In [ ]:
```
"a" > "A"
```

Comparative operators: `in`

`in` is a special operator in Python that checks for "membership".

In [ ]:
```
# is the item present in a list?
1 in [1, 2, 3, 4, 5]
```

In [ ]:
```
# is the item a key of a dictionary?
"apple" in {"apple":0.99, "banana":0.59}
```

In [ ]:
```
# is the item (a string) a substring of a longer string?
"i" in "Team"
```

Logical operators: `and` and `or`

The operators `and` and `or` allow us to ask more sophisticated logical questions.

In [ ]:

```
# <and> returns True if both flanking statements are True
1 < 10 and 10 < 1
```

In [ ]:

```
1 < 10 and 10 > 1
```

In [ ]:

```
# <or> returns True if at least one flanking statement is True
1 < 10 or 1 > 10
```

Logical operators: `not`

`not` *negates* (flips) the truth value that follows it (the logical equivalent of multiplying by -1).

In [ ]:
```
not True
```

In [ ]:
```
not 10 < 1
```

In [ ]:
```
not 1 < 10 and 10 < 1
```

In [ ]:
```
# use parentheses to make the order of execution more explicit
not (1 < 10 and 10 < 1)
```

Expanded operator precedence

- Higher rows have higher precedence (evaluate first)
  - `( )`
  - `[]` (indexing and slicing), `f( )` (function calls)
  - `**`
  - `*`, `/`, `//`, `%`
  - `+`, `-`
  - `in`, <, <=, >, >=, `!=`, ==
  - `not`
  - `and`
  - `or`
- Operators on the same row are tied and evaluate left to right
- When in doubt, add `()`s

Conditionals: the `if` statement

Like the `for` loop, the `if` statement is another common "structure" for building programs. An `if` block will only execute if a given *condition* is `True`.

In [ ]:
```
x = 4
if x % 2 == 0:
    print( x, "is even" )
```

In [ ]:
```
TEST = "Eric" in "America" # <- Boolean value stored in all-caps variable
if TEST:
    print( "I found a substring!" )
```

In [ ]:

```
IS_COLD = True
GOING_OUTSIDE = True
if IS_COLD and GOING_OUTSIDE:
    print( "Put on a jacket!" )
```

Conditionals: the `if/else` statement

The `if/else` statement is slightly fancier: it executes the `if` block if a given condition is `True`, otherwise it executes the `else` block.

In [ ]:

```
x = 4
if x % 2 == 0:
    print( x, "is even" )
else:
    print( x, "is odd" )
```

- `if/else` statements are fundamental to decision making in programs (and life).

In [ ]:

```
traffic_signal = "Red"
if traffic_signal == "Green":
    print( "Let's go!" )
else:
    print( "Stop!" )
```

Conditionals: the `if/elif/else` statement

The `if/elif/else` statement is the most flexible: it allows us to check a variety of possible conditions. Only the block associated with the first `True` condition will be executed. Here, `else` is often used to catch an unexpected option.

In [ ]:

```
traffic_signal = "Yellow"
if traffic_signal == "Green":
    print( "Let's go!" )
elif traffic_signal == "Yellow":
    print( "Prepare to stop." )
elif traffic_signal == "Red":
    print( "Stop!" )
else:
    print( "Unknown signal" )
```

`if`/`elif` differs from a pair of `if` statements:

In [ ]:
```
x = 5
if x > 3:
    print( x, "is greater than 3" )
elif x > 1:
    print( x, "is greater than 1" )
```

In [ ]:
```
if x > 3:
    print( x, "is greater than 3" )
if x > 1:
    print( x, "is greater than 1" )
```

Conditionals in loops

Conditionals frequently arise within loops. There, they allow us to perform different actions depending on the current value of the loop variable. Note the second level of indentation for the `if/else` blocks.

In [ ]:

```
for n in [1, 2, 3, 4, 5]:
    if n % 2 == 0:
        print( n, "is even" )
    else:
        print( n, "is odd" )
```

Conditionals in loops: Fizz Buzz

- "Fizz Buzz" is a children's game in which players count in a circle.
- When it's time to say a number that is divisible by 3, you say "Fizz" instead of the number.
- When it's time to say a number that is divisible by 5, you say "Buzz".
- If the number is divisible by both 3 and 5, you say "Fizz Buzz".

In [ ]:
```
for n in range( 1, 35 ):   # <- range( X, Y ) starts from X and stops at Y
    say = n
    if n % 3 == 0 and n % 5 == 0:
        say = "Fizz Buzz"
    elif n % 3 == 0:
        say = "Fizz"
    elif n % 5 == 0:
        say = "Buzz"
    print( say, end=", " ) # <- terminate with ", " rather than "\n" (newline)
```

Conditionals in loops: Fizz Buzz

The order of the tests in our `if/elif/else` statement different from my description of the game. What happens if I use the original order?

In [ ]:

```
for n in range( 1, 35 ):
    say = n
    if n % 3 == 0:
        say = "Fizz"
    elif n % 5 == 0:
        say = "Buzz"
    elif n % 3 == 0 and n % 5 == 0:
        say = "Fizz Buzz"
    print( say, end=", " )
```

- **Structure conditionals from more to less specific.**

Conditionals in loops: Fizz Buzz

We can also approach this problem with **nested conditionals**:

In [ ]:

```
for i in range( 1, 35 ):
    say = i
    if i % 3 == 0:
        if i % 5 == 0:
            say = "Fizz Buzz"
        else:
            say = "Fizz"
    elif i % 5 == 0:
        say = "Buzz"
    print( say, end=", " )
```

Deeply nested code is harder for PEOPLE to read. Avoid when possible.

Conditionals in loops: Max Price

Find the most expensive fruit in this dictionary of prices:

In [ ]:
```
prices = {
    "apple":      0.99,
    "banana":     0.59,
    "cantaloupe": 2.99,
    "grape":      0.05,
}
```

In [ ]:
```
# a common "motif" for finding a max
max_price = 0
for fruit in prices:
    my_price = prices[fruit]
    if my_price > max_price:
        max_price = my_price
print( max_price )
```

`break` and `continue` change loop behavior

- Executing `break` exits the loop immediately.
- Executing `continue` moves immediately to the next cycle of the loop.

In [ ]:
```
for n in range( 10 ):
    print( n, end=" " )
```

In [ ]:
```
for n in range( 10 ):
    if n > 5:
        break
    print( n, end=" ")
```

In [ ]:
```
for n in range( 10 ):
    if n < 5:
        continue
    print( n, end=" " )
```

The `while` loop

The `while` continues looping as long as a condition is `True`.

In [ ]:
```
x = 0
while x < 10:
    print( x, end=" " )
    x += 1
```

- **If we comment out the `x += 1` line, then `x < 10` will ALWAYS be `True`, and we will loop forever.**
- This is an example of an "infinite loop".
- If your code is "hanging" (running for a long time without doing anything), check for bad `while` loops.

Practice: Logical Operators

In [ ]:
```
# (1) write a logical statement involving numbers that returns True
```

In [ ]:
```
# (2) write a logical statement involving strings that returns False
```

In [ ]:
```
# (3) write a logical statement involving a collection
```

In [ ]:
```
# (4) write a logical statement with two <ands> and one <or> that returns True
```

In [ ]:
```
# (5) write a logical statement with an <and>, <or>, and <not> that returns Fals
e
```

Practice: Conditionals

In [ ]:

```
time = 0
location = "I don't know"
# (1) write an <if/elif/else> block here that determines where you are today bas
ed on the hour (out of 24)
print( location )
```

In [ ]:

```
# (2) expand the block below to offer guesses for the other three combinations o
f the two TESTS
CAN_FLY = True
BIGGER_THAN_BREADBOX = True
if CAN_FLY and BIGGER_THAN_BREADBOX:
    print( "could be an albatross?" )
```

Practice: Conditionals in loops

In [ ]:

```
# (1) modify the loop to find the "earliest" character in the given string (base
d on sorting order)
text = "alphAbet"
earliest = "z" # <- initialized as the last possible character, lower-case z
for char in text:
    continue
print( earliest )
```

In [ ]:

```
# this code uses the <is None> motif to initialize <earliest>
# this way we don't have to think about how to initialize <earliest> in a logica
l way
# (2) What happens if you evaluate this without modifying it?
# (3) What if you copy-paste your loop code from above and then evaluate?
text = "alphAbet"
earliest = None
for char in text:
    if earliest is None:
        earliest = char
    continue
print( earliest )
```

In [ ]:

```
# (4) modify this code to print the fruit with the max price, rather than the pr
ice itself
prices = {
    "apple":      0.99,
    "banana":     0.59,
    "cantaloupe": 2.99,
    "grape":      0.05,
}
max_price = 0
for fruit in prices:
    my_price = prices[fruit]
    if my_price > max_price:
        max_price = my_price
print( max_price )
```

In [ ]:

```
# (5) Modify the "Fizz Buzz" definition below to produce the same output using a
 <while> loop
for n in range( 1, 35 ):
    say = n
    if n % 3 == 0:
        say = "Fizz"
    elif n % 5 == 0:
        say = "Buzz"
    elif n % 3 == 0 and n % 5 == 0:
        say = "Fizz Buzz"
    print( say, end=", " )
```

In [ ]:

```
# (6) CHALLENGE: Implement the "Fizz Buzz" game using three <if> statements with
in a <for> loop
# HINT: You will "build" your saying rather than choosing it
```

Bonus: the *ternary* operator

Simple `if/else` statements (i.e. those with one "line" per block) can be expressed with the **ternary operator** `A if B else C`. This operator returns A if B is `True`, otherwise it returns C.

In [ ]:

```
pattern = "fun"
text = "fundamentals"
answer = "found" if (pattern in text) else "missing"
print( answer )
```