Conditionals and Flow of Control

Eric Franzosa, Ph.D. (franzosa@hsph.harvard.edu)
2018-09-18

Outline

- HW2 + office hours notes
- Review of key concepts from week 1
- Boolean values
- Comparative and logical operators
- Conditionals: `if`/`elif`/`else`
- The `while` loop and loop control
- Practice

Review: Variables

- A variable is a "bucket" for storing data (or a reference to data).
- Acting on a variable is the same as acting on the data it contains (or refers to).

In [ ]:
```
"hello, world!".upper( )
```

In [ ]:
```
message = "hello, world!"
message.upper( )
```

Review: Scalar Data

We'll use four `types` of **scalar** data in this course

- `str`ings
- `int`s (integers)
- `float`s (decimal numbers)
- `bool`ean values (more on those today)

Scalar data are **immutable**: operations on the data don't change the data.

In [ ]:
```
x = 5
x + 5
print( x )
```

Scalar data are "updated" by overwriting the variables that contain them.

In [ ]:
```
y = 5
y = y + 5
print( y )
```

Review: Collections

We'll use three main data **collections** in this course:

- `lists` are ordered collections of elements
- `dicts` (dictionaries) are mappings fr om one type of data to another
- `sets` are unordered collections of unique elements

Collections are **mutable**: we can change them after they are created.

In [ ]:
```
x = [0,1,2,3,4]
print( x )
```

In [ ]:
```
x[0] = "Hello"
x.append( "World" )
print( x )
```

Review: Transformations of Data

In Python we can transform data using **functions**, **methods**, and **operators**.

In [ ]:
```
# functions take data as arguments and return some output
sum( [1, 2, 3, 4, 5] )
```

In [ ]:
```
# methods are functions associated with particular data
"hello, world!".upper( )
```

The piece of data that "owns" acts as an implied first argument: in other words, `x.method( )` behaves similarly to `method( x )`.

Operators are usually symbols and act on the data surrounding them (called *operands*).

In [ ]:
```
5 + 5
```

The same operator can do different things depending on the `types` of the surrounding data:

In [ ]:
```
"5" + "5"
```

In [ ]:
```
[5] + [5]
```

Review: the `for` loop

The `for` loop allows us to repeat a block of code. Properties of the `for` loop:

- Definition of one (or more) temporary variables (e.g. x below)
- Which store the elements of an iter able piece of data (e.g. a list of numbers)
- Which are manipulated in an indented block of code

In [ ]:
```
for x in [0,1,2,3,4]:
    x = x + 1
    # note: a new aspect of print( ), ending with " " rather than a new line
    print( x, end=" " )
```

Outline

- ~~HW2 + office hours notes~~
- ~~Review of key concepts from week 1~~
- 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`. They are used by computers to track objectively true and false statements within code.

"The blue whale is the largest mammal": **Objectively true.**

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

"The grizzly bear is the best type of bear": **An 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 alphabetical 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?
"Eric" in "America"
```

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 < 100
```

In [ ]:
```
1 < 10 and 10 > 100
```

In [ ]:
```
# <or> returns True if at least one flanking statement is True
1 < 10 or 10 > 100
```

Logical operators: `not`

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

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

In [ ]:
```
not 100 > 10
```

In [ ]:
```
not 2 < 10 or 10 > 100
```

In [ ]:
```
# use parentheses to make the order of execution more explicit
not (2 < 10 or 10 > 100)
```

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"
if TEST:
    print( "I found a substring!" )
```

Recall an example from an earlier lecture:

In [ ]:
```
RAINING = True
DAYTIME = True
if RAINING and DAYTIME:
    print( "I went to the movies"  )
```

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" )
```

Conditionals: the `if/else` statement

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

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

Conditionals: 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 )
```

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( "Slow down, prepare to stop."  )
elif traffic_signal == "Red":
    print( "Stop!" )
else:
    print( "Unknown signal; proceed with caution."  )
```

`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, the y allow us to perform different actions depending on the current value of the loop variable. Note the second le vel of indentation for the `if/else` blocks.

In [ ]:
```
for i in [1,2,3,4,5]:
    if i % 2 == 0:
        print( "Even", end=" " )
    else:
        print( "Odd", end=" " )
```

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

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 i in range( 1, 35 ):
    say = i
    if i % 3 == 0:
        say = "Fizz"
    elif i % 5 == 0:
        say = "Buzz"
    elif i % 3 == 0 and i % 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. A void 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 i in range( 10 ):
    print( i, end=" " )
```

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

In [ ]:
```
for i in range( 10 ):
    if i < 5:
        continue
    print( i, 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: The Collatz Conjecture

Consider the following algorithm that acts on a positiv e interger **n**:

- If **n** is even, divide **n** by 2, then repeat this process.
- If **n** is odd, triple **n** and add 1, then repeat this process.

No matter what **n** we start with, if we repeat the abo ve rules over and over, we always seem to end up at 1. Indeed, The Collatz Conjecture (https://en.wikipedia.org/wiki/Collatz_conjecture) is that all numbers will end up at 1: a fact that has never been proven (but no counterexamples have been found).

- **Convince yourself**: Pick a number and verify that repeating the above process leads to 1.
- For example, if I start with **n** = 12, I pass through **n** = 6, 3, 10, 5, 16, 8, 4, 2, and finally 1.

Practice: The Collatz Conjecture

Below I've written a Python function to generate a Collatz "chain". Try to describe in words what the function is doing using concepts from toda y's lecture. (`//` is the *integer division operator*: it returns the integer part of a quotient between two numbers, whereas normal division, `/`, always returns a decimal answer.)

In [ ]:
```
def collatz( n ):
    chain = [n]
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        chain.append( n )
    return chain
```

In [ ]:
```
# try experimenting with different numbers
collatz( 12 )
```

What would happen if we called `collatz( )` with an argument (number) that violated the Collatz Conjecture? In other words, a (theoretical) number whose chain never arrived at 1?

Practice: The Collatz Conjecture: Exercise 1

- Write a Python loop to try to find the longest Collatz chain between 2 and 100.
- What number produces the chain?
- How long is the chain?

In [ ]:

```
# here is a code "skeleton" to get you started
best_n = 1
best_chain = []
# replace [] to loop through the numbers 2-100
for n in []:
    # replace [] with a function call to get n's chain
    my_chain = []
    # replace 0 to test if this is the longest chain we've seen
    if len( my_chain ) > 0:
        # replace 1 and [] to track the best n and its chain
        best_n = 1
        best_chain = []

# results
print( best_n )
print( len( best_chain ) )
print( best_chain )
```

Practice: The Collatz Conjecture: Exercise 2

Write Python code to determine how man y Collatz chains (up to a given number) pass through a given number. For example, there are 5 chains starting between 2 and 100 that pass through 25.

In [ ]:
```
# your code here
```

Practice: The Collatz Conjecture: Exercise 3

Write Python code to determine the most frequently visited numbers in chains starting between 2 and 100. Note that we can use a dictionary in this process:

In [ ]:
```
counts = {}
for n in [9, 99, 99, 999, 999, 999]:
    # if n is not already in counts, we will get an error when we try to inc
rease it
    if not (n in counts):
        counts[n] = 0
    counts[n] = counts[n] + 1
print( counts )
```

In [ ]:
```
# your code here
```

More like this: Project Euler

If you like math puzzles like the ones above, Project Euler (https://projecteuler.net/) is full of them. It is also a great place to pr actice a new programming language. Providing the correct answer to a problem unlocks the ne xt (slightly harder) problem.