Python Conditionals

Python has conditionals that return boolean values. Python supports `==` for "equal" and `!=` for "not equal". Comparisons also work for strings and a number of other types *for which it makes sense.*

```
>>> 0 == 1
False
>>> 0 == 0
True
>>> 0 != 1
True
>>> "a" == "A"
False
>>> "a" == "a"
True
```

Other comparisons such as "less than" or "greater than or equal", etc., work as you would expect. Note that comparison of strings happens in lexicographical order, so capital letters are "less than" lowercase letters.

```
>>> 1 > 0
True
>>> 2 >= 3
False
>>> -1 < 0
True
>>> .5 <= 1
True
>>> "a" < "b"
True
>>> "apple" < "animal"
False
>>> "Apple" < "apple"
True
>>> 'Z' < 'a'
True
```

One thing you might not expect is the ability to chain comparisons, unlike languages such as C++ or Java.

```
>>> x = 5
>>> y = 4
>>> 1 < y < x
True
```

Controlling code paths with `if` statements and conditional expressions

When you need to execute code conditionally, Python uses `if` statements that use conditional expressions like what is shown above. Note that the line with the `if` statement ends with a `:` - this is required and tells Python that it should expect the code following it to be indented.

```
>>> x = 6
>>> y = 5
>>> if x > y:
...     print('x is greater than y!')
...
x is greater than y!
```

The statement on the line after the `if` only gets executed if the conditional expression is `True`. The line **must** be indented. Your editor should let you use the `tab` key to insert 4 spaces. Note the REPL prompt changes to the `...`. We can continue to add lines of code to this "code block". Type an extra return/enter to end the block.

Python is very particular about indentation. Languages like C++ and Java use braces or curly brackets (`{}`) to delimit code blocks. Python doesn't work like that; it uses the indentation level to determine the code block. This keeps the code less cluttered.

All code blocks must be indented and the indentation must be the same on each line, when there are multiple lines in the code block. The convention is to use 4 spaces per indentation level, so that is what I expect you to use.

When we want code to execute when the conditional expression of the `if` statement is `False`, we use an `else` clause.

```
>>> score = 82
>>> if score >= 70:
...     grade = 'Pass'
... else:
...     grade = 'Fail'
...
>>> grade
'Pass'
```

What happens if you have more things to test instead of just using an `else` clause? Python uses `elif` as a contraction of "else if", to continue more testing.

```
>>> if score >= 90:
...     grade = 'A'
... elif score >= 80:
...     grade = 'B'
... elif score >= 70:
...     grade = 'C'
... elif score >= 60:
...     grade = 'D'
... else:
...     grade = 'F'
...
>>> grade
'B'
```

The final `else` is not required, whether or not you have any `elif`, and no matter how many `elif` statements there are. However, if it is there, the `else` is always last!

De Morgan's Laws

Python has boolean logical operators `and` and `or`, plus `not` for negation. We can also use the operators `in` and `not in` to test containment:

```
>>> 1 < 2 and 'x' in 'abc'
False
>>> 1 < 2 or 'x' in 'abc'
True
>>> not (1 < 2)
False
>>> 'a' not in 'abcde'
False
```

De Morgan's Laws come from propositional logic and boolean algebra. But don't let that scare you. It is simply the formalized relationship between `and`, `or`, and `not`. If we have boolean expressions `E1` and `E2` and another expression (`E1 and E2`), then by using De Morgan's laws, we can express the *negation* of (`E1 and E2`) as (`not E1 or not E2`). Similarly, the *negation* of `E1 or E2` can also be expressed as `not E1 and not E2`. Sometimes this comes in handy when we want to write these expressions in code, because it might be easier or cleaner to express it one way vs. another.

```
>>> x = 4
>>> y = 7
>>> x < 5 and y > 5
True
>>> not (x < 5 and y > 5)
False
>>> not x < 5 or not y > 5
False
```

Short-circuiting of logical expressions

Like other languages, Python uses what we call *short-circuiting* in evaluating logical expressions. When we have an expression using `and`, if the first part of the expression resolves to `False`, then Python does not evaluate the rest of the expression, because it knows the answer will be `False`, no matter what the other expression resolves to.

Similarly, if the first part of an expression using `or` is `True`, then the remainder of the expression is not evaluated, as the answer will be `True`, no matter what the other expression resolves to.

Other Operators

Python has bitwise operators: `&` for bitwise 'and', `|` for bitwise 'or', and `^` for bitwise 'xor'. In addition, there are `>>` and `<<` for bitwise shifting. These can only be used on integers.

```
>>> 4 << 2
16
>>> 4 | 2
6
>>> 4 & 2
0
>>> 6 & 3
2
>>> 7 ^ 4
3
>>>
```

Modulo operator

Somehow I left off the **modulo** (`%`) operator in the discussion of numeric operators in the lecture. It is a binary operator and returns the *remainder* of one number divided by another.

```
>>> 7 % 3
1
>>> 123 % 11
2
>>> 123 % 2
1
```

The modulo operator is useful when you want to know if a number is *evenly divisible* by another number. For example, if you want to know if a number is an even number, you can tell by checking that the number modulo 2 is zero. Conversely, odd numbers are not evenly divisible by 2, so the result of an odd number modulo 2 will be 1.

```
>>> def is_even_number(number):
...     return number % 2 == 0
...
>>> def is_odd_number(number):
...     return number % 2 == 1
...
>>> is_even_number(3)
False
>>> is_even_number(6)
True
>>> is_even_number(189743)
False
>>> is_odd_number(189743)
True
>>>
```

You can take advantage of Python's **truthiness** of numbers when you're inlining tests of whether numbers are evenly divisible or not. A number that is zero will evaluate as `False`, while any other value will evaluate to `True`. So if the result of `number % 2` is `True`, then the number is odd.

```
>>> def print_odd_or_even(number):
...     if number % 2:
...         print("number is odd")
...     else:
...         print("number is even")
...
>>> print_odd_or_even(7866)
number is even
>>> print_odd_or_even(7)
number is odd
>>> print_odd_or_even(0)
number is even
```

In the statement `if number % 2`, python evaluates the truthiness of the result of the expression `number % 2`. If the result is not zero (truthy), then the number is odd.