If and Comparisons

The if-statement uses comparisons like `i < 6` to control if lines run or not.

If Statement

The if-statement controls if some lines run or not. A "boolean" is a value which is True or False. The if-statement has a boolean-test, a colon, and indented lines of code (similar to "while"):

```
if boolean-test:
  indented body lines
```

The if-statement first evaluates the the boolean test, and if it is True, runs the "body" lines once. Otherwise if the test is False, the body lines are skipped and the run continues after the last body line.

The simplest and most common sort of boolean test uses == (two equal signs next to each other) to compare two values, yielding True if the two are the same.

Here is an example that shows an if-statement inside a for-loop. Every time through the loop, the test `num == 6` is evaluated, yielding boolean True or False each time.

```
>>> for num in [2, 4, 6, 8]:
        if num == 6:
            print('Here comes 6!')
        print(num)
2
4
Here comes 6!
6
8
```

If test = vs. == Syntax Error

It's very easy to accidentally type a single equal sign for a comparison like the following, but in Python that is flagged as a syntax error:

```
if num = 6:        # typo, meant ==
    print('hi')
```

Style: Do Not Write x == True

Suppose some foo() function is supposed to return True or False. Do not write an if-test like this:

```
if foo() == True:   # NO, do not use == True
    print('yay')
```

Instead, let the if/while take the boolean value directly like this:

```
if foo():           # YES this way
    print('yay')


# Or to test if foo() is False use not:
if not foo():
    print('yay is cancelled')
```

if else:

The optional `else:` part of an if-statement adds code to run in the case that the test is False. Use `else:` if the run should choose between either action-1 or action-2 depending on the test, but not do nothing. Use regular if to do action-1 or nothing.

```
# set message according to score
if score > high_score:
    message = 'New high score!'
else:
    message = 'Oh well!
```

To run code if a test is False and otherwise do nothing, use `not` like this:

```
if not foo():
    message = 'no foo today'
```

Boolean Comparison Operators

The most common way to get a boolean True/False is comparing two values, e.g. the comparison expression `num == 6` evaluates to True when num is 6 and False otherwise.

Comparison operators:

== test if two values are equal (2 equals signs together). Works for int, string, list, dict, .. most types. Not recommended to use with float values.

!= not-equal, the opposite of equal (uses == under the hood)

< <= less-than, less-or-equal. These work for most types that have an ordering: numbers, strings, dates. For strings, < provides alphabetic ordering. Uppercase letters are ordered before lowercase. It is generally an error to compare different types for less-than, e.g. this is an error: `4 < 'hello'`

> >= greater than, greater-or-equal.

The interpreter examples below shows various == style comparisons and their boolean results:

```
>>> 5 == 6
False
>>> 5 == 5
True
>>> 5 != 6
True
>>> 4 > 2
True
>>> 4 > 5
False
>>> 4 > 4
False
>>> 4 >= 4  # contrast >= vs. >
True
>>> s = 'he' + 'llo'
>>> s == 'hello'
True
>>> 'apple' < 'banana'
True
>>> 'apple' < 4
TypeError: '<' not supported between instances of
'str' and 'int'
```

if elif:

There is a more rarely used `elif` for where a series of if-tests can be strung together (mnemonic: 'elif' is length 4, like 'else'):

```
if s == 'a':
    # a case
```

```
elif s == 'b':
    # b case
else:
    # catch-all case
```

The tests are run top to bottom, running the code for the first that is True. However, the logic of when each case runs can be hard to see. What must be true for case c below? You really have to think about the code work work out when (c) happens.

```
if score > high and s != 'alice':
    # a
elif s == 'bob':
    # b
else:
    # c
```

Answer: c happens when s is not 'bob' but also (score <= high or s == 'alice or both)

If/else chains are fine, just don't think they are trivial. Only add `else` if the code needs it. Nice, simple if handles most problems and is the most readable.

Return vs. if/else chains

Nick style idea: only use `else` when it is truly needed. If the code can work using only a plain-if, I prefer to write it that way. This works extra well with decomposed out functions, where 'return' can be used to bail out for the first few cases, leaving the main case below without indentation, like this:

```
def foo(s):
    if len(s) < 4:
        return 'too small!'

    if len(s) > 100:
        return 'too big!'

    # rest of computation having screened out too-
short and too-long
    # notice: no if-else structure, not indented down
here
    ...
```

You can use else if you prefer, just thinking about possible alternative structure here.

Truthy True/False

We think of if/while tests as looking at boolean values, however the rules are flexible so any type can work in there.

What does this code do?

```
s = 'hello'
if s:
    print('truthy')
```

What does `s` mean as an if-test? Python has rules for this we'll call "truthy" logic.

Truthy Logic

The truthy rules define a series of values which count as `False`. All the sort of "empty" values count as `False`: **`0, 0.0, None, '', [], {}`**

When an if-test expression is something like `0` or `None` or the empty string `''`, that counts as `False`. Any other values counts as `True`. The int `6` or the non-empty string `'hi'`, or the list `[1, 2]` all count as `True`.

The `bool()` function takes any value and returns a formal bool `False/True` value, so it's a way for us to see how truthy logic works in the interpreter:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')    # empty string - False
False
>>> bool([])    # empty list - False
False
>>> bool(None)
False
>>> bool(6)         # non-zero int - True
True
>>> bool('hi')      # non-empty string - True
True
>>> bool([1, 2])    # list of something - True
True
>>> bool('False')   # tricky: what's this one?
???
```

Truthy Shorthand

Why does the truthy system exist? It makes it easy to test, for example, for an empty string like the following. Testing for "empty" data is such a common case, it's nice to have a shorthand for it. For CS106A, you don't ever need to use this shorthand, but it's there if you want to use it.

```
# long form screen out empty string
if len(s) > 0:
    print(s)


# shorter way, handy!
if s:
    print(s)
```