Code Documentation

But first, a word about strings

In the week 1 lectures, I forgot to include a few special case things about Python's string literals. First of all, strings in Python are by default Unicode, unlike Python 2 where you needed to specify a unicode string as `u'text in unicode'`. Encodings are beyond the scope of the class; if you are interested, you can find a lot of material on the internet. Be sure you understand the Python version that applies to whatever article, blog, etc., that you are reading.

If we want to include backslashes in a string to escape something, Python understands that. For example, making a multi-line string without the triple-quotes:

```
>>> my_string = 'this is a line\nfollowed by another'
>>> my_string
'this is a line\nfollowed by another'
>>> print(my_string)
this is a line
followed by another
```

We can see that when we print a string, Python interprets the escape sequence `\n` to be an end-of-line, and displays our string on two lines.

If we have a string with a lot of backslashes, or where it would be confusing/awkward to have to go in and "escape" the backslash, we can use *raw strings*, denoted by prefixing with `r` or `R`. Python ignores escape sequences in raw strings.

```
>>> r_string = r"We use '\n' to indicate a newline"
>>> r_string
"We use '\\n' to indicate a newline"
>>> print(r_string)
We use '\n' to indicate a newline
```

Python also has *bytes* string literals, which are created by prefixing the string with a `b` or `B`. These can only have ASCII characters in them, and if you want bytes above 127, you have to represent them with escape sequences. You can do most things with bytes as with other strings, but you can't mix string types:

```
>>> b_string = b'This is a bytestring'
>>> b_string.upper()
b'THIS IS A BYTESTRING'
>>> b_string.split()
[b'This', b'is', b'a', b'bytestring']
>>> b_string + b'.'
b'This is a bytestring.'
>>> b_string + '.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat str to bytes
>>> b_string + b'.'
b'This is a bytestring.'
```

Raw Strings and Regular Expressions

You can ignore this part about regular expressions if you don't use them, and skip to Code Documentation. Raw strings are especially useful when putting together regular expressions for search strings. As a simple example, say we

want to search for a whitespace character followed by 3 digits, followed by another whitespace character. Our search string would be `'\s\d{3}\s'`. If we didn't have raw strings, we would have to use `'\\s\\d{3}\\s'` for our search string. Yikes, as if regex expressions aren't hard enough to read! But, we can use raw strings and still be able to read our regex expression.

For anyone interested in doing regex in Python, here is how that would work, using the builtin `re` module:

```
>>> import re
>>> my_string = 'There are over 300 different dog breeds'
>>> re.search(r'\s\d{3}\s', my_string)
<_sre.SRE_Match object; span=(14, 19), match=' 300 '>
```

Regular expressions are not part of this course - it would take several weeks to do it justice in Python, but you're welcome to experiment to learn more. I know some of you do testing and write code to analyze log file data, which often involves a lot of searching with regular expressions. Just remember, whenever you use the `re` module, make it a habit to always use raw strings even when there are no backslashes. It's just a good habit to get into with regular expressions.

You won't need to know any regular expressions for this class or for the homework. If you turn in homework using `re`, I will know you did not write the homework yourself.

And now, back to Code Documentation

Comments. The purpose of comments is to make the code easier for humans to understand. Comments begin with a hash symbol (pound sign, number sign, octothorpe): `#`.

A comment should describe **why** the code is doing what it's doing. Don't describe **what** it is doing as that is generally obvious. Or at least it should be. Obfuscation of code is frowned upon in Python. To paraphrase a famous quote, you should code as if the person who will maintain your code has anger-management issues and *knows where you live*.

Python does not have multi-line comments. Each line must have a `#` symbol, and indentation alignment needs to be preserved. Most editors have a way to comment or un-comment multiple selected lines. In Atom, it is `CTRL-/` or `CMD-/` on a Mac.

Docstrings. A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition to describe it:

```
def adder(a, b):
    """ Return the result of a + b. """
    return a + b
```

We use `"""triple double quotes"""` (multi-line) strings for docstrings, *even when they are only one line*. This is for clarity and consistency, as triple-quoted strings stand out more. Use `r"""raw triple double quotes"""` if you use any backslashes in your docstrings.

If we had more information to put in the docstring, it would be multiple lines, and we are already prepared with the triple-quoted string:

```
def do_a_thing(arg1, arg2):
    """ Summary line of do_a_thing.

        Detailed description of do_a_thing goes here.
        This function does a thing that we are describing in detail here.
```

```
        arg1: Description of arg1; include type only if meaningful
        arg2: Description of arg2; include type only if meaningful
        Returns: Description of what it returns
        Raises: Description of special error messages raised in do_a_thing
    """
```

You will find that I do not use this multi-line format for our homework assignments, mainly because it is overkill for the simple code we do. However, in real life, I do follow it. Typically, every company has its own conventions for documentation, so you should just follow that.

Note that a triple-quoted string is not a comment, it is actual code. It is just like a regular string (delimited by a single or double quote), except we can span multiple lines if we want. Using them for docstrings (instead of comments) is done on purpose, as there are features of Python that use docstrings internally, whereas comments are stripped out when the code is executed. Later we will see how the docstring works in a command-line program.

Do not use triple-quoted strings as comments in your code! They are only interpreted as docstrings when they appear at the beginning of modules, classes or functions/methods. Use `#` prefixes for code comments.

If you have any questions, please ask!

