We continue experimenting with simple class design.

1. Questions?

2. How to determine if a function we're building should print, return, yield, or None.
   
   (a) print only if the docstring, instructions say to “print”.
   
   (b) yield is often indicated by “generate” or instructions stating it’s a “generator” or “iterator"
   
   (c) return can be indicated by “return”, but not always! yield and return functions produce a
       value that can be stored in a variable when it's called (i.e., newlist = sorted(mylist))
   
   (d) None are functions that modify a variable, object state, etc. without a need to return
   
   (e) Apply logic from the program design when considering which of these to choose!

3. Functions versus Methods

4. Naming conventions

   (a) Classes with an uppercase, variableName with a lowercase (camelCase).

   (b) __xxxx__(..) are special python functions, variables

   (c) _objectName are functions, variables we don't want externally accessed

   (d) _ is an object name, for objects we never refer to again. Used sometimes in for..loops.

5. The Pt class.

6. Thinking about Ratios of integers.

   (a) If we have gcd of two values, a and b we can compute the greatest common divisor with code
       similar to this:

       ```python
       def gcd(a,b):
           while a != 0:
               (a,b) = (b%a,a)
           return b if b >= 0 else -b
       ```

   (b) Again: the __slots__ attribute of a class pre-declares the attributes of individual objects con-
       structed by the class. You cannot add any attributes that are not mentioned in the __slots__
       list. For ratios, perhaps we'd have:

       ```python
       __slots__ = ['_top', '_bottom']
       ```
(c) Annotations. Python provides a rich collection of syntactic notes that can change how code is interpreted, called annotations. These are typically prefixed with the at-sign (@).

(d) We learned that we can write accessor methods for our classes. If we would like to treat those accessors like attributes, we can use the @property annotation:

```python
@property
def numerator(self):
    return self._top
```

Given this, we’re now able to write

```python
r = Ratio(10,15)
print("Numerator is {}".format( r.numerator ))
```

Note the missing parentheses! You cannot, however, assign a value to r.numerator—it’s read-only.

(e) If you do want to be able to set this pseudo attribute, you can declare a setter:

```python
@numerator.setter
def numerator(self,value):
    self._top = value
```

(f) Where meaningful, we can overload the meaning of arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td><strong>eq</strong></td>
<td>Test for equality</td>
</tr>
<tr>
<td>&lt;</td>
<td><strong>lt</strong></td>
<td>Test for less</td>
</tr>
<tr>
<td>-a</td>
<td><strong>neg</strong></td>
<td>Negation operator</td>
</tr>
<tr>
<td>+a</td>
<td><strong>pos</strong></td>
<td>Positive operator</td>
</tr>
<tr>
<td>+</td>
<td><strong>add</strong></td>
<td>Sum of values</td>
</tr>
<tr>
<td>-</td>
<td><strong>sub</strong></td>
<td>Difference of values</td>
</tr>
<tr>
<td>*</td>
<td><strong>mul</strong></td>
<td>Product of values</td>
</tr>
<tr>
<td>/</td>
<td><em>truediv</em></td>
<td>Ratio of two values</td>
</tr>
<tr>
<td>%</td>
<td><strong>mod</strong></td>
<td>Remainder after division</td>
</tr>
<tr>
<td>//</td>
<td><strong>floordiv</strong></td>
<td>Whole division</td>
</tr>
</tbody>
</table>

The class annotation @total_ordering, imported from functools, will generate all comparison operations from __lt__ and __eq__.

(g) Where common operators are not implemented, we return NotImplemented.

(h) The __str__ implements the str(r) printable string method

(i) The __repr__ implements the repr(r) representation string method

(j) Many more! See the documentation:

https://docs.python.org/2/reference/datamodel.html#emulating-numeric-types

*