Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hop Contest entries due Thursday 10/3 @ 11:59pm
- Composition scores will be assigned this week (perhaps by Monday).
- 3/3 is very rare on the first project.
- You can gain back any points you lose on the first project by revising it (November).

Data

Data Types

Every value has a type

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>There are primitive expressions that evaluate to values of these types.</td>
</tr>
<tr>
<td>2.</td>
<td>There are built-in functions, operators, and methods to manipulate those values.</td>
</tr>
</tbody>
</table>

Numeric types in Python:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt; type(2)</td>
<td>int</td>
</tr>
<tr>
<td>&gt;&gt;&gt; type(1.5)</td>
<td>float</td>
</tr>
<tr>
<td>&gt;&gt;&gt; type(1+1j)</td>
<td>complex</td>
</tr>
</tbody>
</table>

Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create abstractions.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object.
  - All objects have attributes.
  - A lot of data manipulation happens through object methods.
  - Functions do one thing; objects do many related things.

Data Abstraction
Data Abstraction

- Compound objects combine objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- Abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use

Rational Numbers

- Exact representation of fractions
- A pair of integers
- As soon as division occurs, the exact representation may be lost!
- Assume we can compose and decompose rational numbers:

```
def rational(n, d):
    return rational(numer(n), denom(d))
def numer(x):
    return numer(n)
def denom(x):
    return denom(d)
```

Rational Number Arithmetic

```
3/2 + 3/5 = 9/10
3/2 + 3/5 = 21/10
```

Rational Number Arithmetic Implementation

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)
def equal_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Pairs as Tuples

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import itemgetter
>>> itemgetter(pair, 0)
1
>>> itemgetter(pair, 1)
2
```

More tuples next lecture
Representing Rational Numbers

```python
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem

def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number x."""
    return getitem(x, 1)
```

Reducing to Lowest Terms

```python
from fractions import gcd
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    g = gcd(n, d)
    return (n // g, d // g)
```

Abstraction Barriers

```
<table>
<thead>
<tr>
<th>Rational numbers as whole data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_rational  mul_rational  equal_rational</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rational numbers as numerators &amp; denominators</th>
</tr>
</thead>
<tbody>
<tr>
<td>rational  numer  denom</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rational numbers as tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuple  getitem</td>
</tr>
</tbody>
</table>

However tuples are implemented in Python

Violating Abstraction Barriers

```
def add_rational((1, 2), (1, 4))

def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])
```

Data Representations

```
add_rational( (1, 2), (1, 4) )
```

Abstraction Barriers

```
add_rational  mul_rational  equal_rational
```

```
rational  numer  denom
```

```
tuple  getitem
```

However tuples are implemented in Python
What is Data?

- We need to guarantee that constructor and selector functions work together to specify the right behavior.
- Behavior condition: If we construct rational number $x$ from numerator $n$ and denominator $d$, then $\text{numerator}(x)/\text{denominator}(x)$ must equal $n/d$.

An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).

- If behavior conditions are met, then the representation is valid.

You can recognize abstract data types by their behavior, not by their class.

Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple. But is that the only way to make pairs of values? No!

Constructors, selectors, and behavior conditions:

If a pair $p$ was constructed from elements $x$ and $y$, then:

- $\text{getitem_pair}(p, 0)$ returns $x$, and
- $\text{getitem_pair}(p, 1)$ returns $y$.

Together, selectors are the inverse of the constructor.

Generally true of container types.

Not true for rational numbers because of GCD.

Using a Functionally Implemented Pair

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions.

If a pair $p$ was constructed from elements $x$ and $y$, then:

- $\text{getitem_pair}(p, 0)$ returns $x$, and
- $\text{getitem_pair}(p, 1)$ returns $y$.

This pair representation is valid!