Announcements
- Homework 3 due Wednesday 10/4 @ 11:59pm
- Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/4 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week. 3/3 is unusual on the first project. You can gain back composition points you lost on Project 1 by revising it (in November)
- Midterm 1 should be graded by Friday. Solutions to Midterm 1 will be posted after lecture.
- Guerrilla section this Saturday 12-2 and 2:30-5 on recursion (Please RSVP on Piazza!)
- Practical Programming now meets Wednesdays 6:30-8pm in 485 Soda

Data Types

Every value has a type

<table>
<thead>
<tr>
<th>Properties of native data types:</th>
<th>(demo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. There are primitive expressions that evaluate to values of these types.</td>
<td></td>
</tr>
<tr>
<td>2. There are built-in functions, operators, and methods to manipulate those values.</td>
<td></td>
</tr>
</tbody>
</table>

Numeric Types in Python:

```python
>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>

>>> type(1+1j)
<class 'complex'>
```

Objects

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

Data Abstraction

Compound objects combine objects together

- A date: a year, a month, and a day
- A geographic position: latitude and longitude

An 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

<table>
<thead>
<tr>
<th>numerator</th>
<th>denominator</th>
</tr>
</thead>
</table>

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:

```python
class Rational:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def numerator(self):
        return self.numerator

    def denominator(self):
        return self.denominator
```

Rational Number Arithmetic

\[
\begin{align*}
3 \times 3 &= 9 \\
2 \times 5 &= 10
\end{align*}
\]

Example

\[
\begin{align*}
\frac{nx}{dx} \times \frac{ny}{dy} &= \frac{nx \times ny}{dx \times dy} \\
\frac{nx}{dx} + \frac{ny}{dy} &= \frac{nx \times dy + ny \times dx}{dx \times dy}
\end{align*}
\]

General Form

** Constructors **

- \( \text{rational}(n, d) \) returns a rational number

** Selectors **

- \( \text{numer}(x) \) returns the numerator of \( x \)
- \( \text{denom}(x) \) returns the denominator of \( x \)

---

Rational Number Arithmetic Implementation

```python
def mul_rational(x, y):
    return rational(numer(x) * numer(y), denom(x) * denom(y))

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 print_rational(x):
    print(numer(x), '/', denom(x))

def rationals_are_equal(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

---

Representing Pairs Using Lists

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

More Lists next lecture

---

Reducing to Lowest Terms

** Example:**

\[
\begin{align*}
\frac{3}{5} \times \frac{1}{2} &= \frac{3 \times 1}{5 \times 2} = \frac{3}{10} \\
\frac{2}{3} + \frac{1}{2} &= \frac{2 \times 2 + 1 \times 3}{3 \times 2} = \frac{4 + 3}{6} = \frac{7}{6}
\end{align*}
\]

```
from fractions import gcd

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

---

Abstraction Barriers

<table>
<thead>
<tr>
<th>Parts of the program that...</th>
<th>Treat rationals as...</th>
<th>Using...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use rational numbers to perform computation</td>
<td>whole data values</td>
<td>add_rational, mul_rational, rationals_are_equal, print_rational</td>
</tr>
<tr>
<td>Create rationals or implement rational operations</td>
<td>numerators and denominators</td>
<td>rational, numer, denom</td>
</tr>
<tr>
<td>Implement selectors and constructor for rationals</td>
<td>two-element lists</td>
<td>list literals and element selection</td>
</tr>
</tbody>
</table>

```
Abstraction Barriers

Use rational numbers to perform computation whole data values add_rational, mul_rational, rationals_are_equal, print_rational
Create rationals or implement rational operations numerators and denominators rational, numer, denom
Implement selectors and constructor for rationals two-element lists list literals and element selection
```

---

Pairs

Representing Pairs Using Lists

- A list literal: Comma-separated expressions in brackets
- "Unpacking" a list
- Element selection using the selection operator
- Element selection function

More Lists next lecture

---

Representing Rational Numbers

```python
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]

def numer(x):
    """Return the numerator of rational number X."""
    return x[0]

def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

---

Reducing to Lowest Terms

** Example:**

\[
\begin{align*}
\frac{3}{5} \times \frac{1}{2} &= \frac{3 \times 1}{5 \times 2} = \frac{3}{10} \\
\frac{2}{3} + \frac{1}{2} &= \frac{2 \times 2 + 1 \times 3}{3 \times 2} = \frac{4 + 3}{6} = \frac{7}{6}
\end{align*}
\]

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Violating Abstraction Barriers

- Does not use constructors
- Twice!

```python
def add_rational(a, b):
    return [a[0] + b[0], a[1] + b[1]]
```

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

Data Representations

Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element list. But isn't 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
  - select(p, 0) returns x, and
  - select(p, 1) returns y.

Together, selectors are the inverse of the constructor.

Generally true of container types.

---

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 a and denominator b, then numerator/denominator must equal a/b.
- 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.

---

Functional Pair Implementation

```python
def pair(x, y):
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
        return get
    return get
    Constructor is a higher-order function.

def select(p, i):
    """Return the element at index i of pair p."""
    return p(i)
    Selector defers to the object itself.
```

---

Using a Functionally Implemented Pair

```python
>>> p = pair(2, 3)
>>> select(p, 0)
1
>>> select(p, 1)
2
```

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

---

Interactive Diagram