Announcements

- Project 3 is due Thursday 10/24 @ 11:59pm
- Extra reader office hours this week:
  - Tuesday 6-7:30 in Soda 405
  - Wednesday 5:30-7 in Soda 405
  - Thursday 6:30-7 in Soda 320
- Midterm 2 is on Monday 10/28 7pm-9pm
  - Topics and locations: [http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html](http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm2.html)
  - Emphasis: mutable data, object-oriented programming, recursion, and recursive data
  - Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm!
  - Review session on Saturday 10/26 from 1pm to 4pm in 1 Pimentel
  - HKN review session on Sunday 10/27 from 4pm to 7pm to 2050 VLSB
- Homework 7 is due Tuesday 11/5 @ 11:59pm (Two weeks)
- Respond to lecture questions: [http://goo.gl/FZKvgm](http://goo.gl/FZKvgm)

Generic Functions of Multiple Arguments

**More Generic Functions**

A function might want to operate on multiple data types

*Last time:*
  - Polymorphic functions using message passing
  - Interfaces: collections of messages that have specific behavior conditions
  - Two interchangeable implementations of complex numbers

*Today:*
  - An arithmetic system over related types
  - Type dispatching
  - Data-directed programming
  - Type coercion

*What’s different?* Today’s generic functions apply to multiple arguments that don’t share a common interface.

Rational Numbers

Rational numbers represented as a numerator and denominator

class Rational:
    def __init__(self, numer, denom):
        g = gcd(numer, denom)  # Greatest common divisor
        self.numer = numer // g
        self.denom = denom // g

    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numer, self.denom)

def add_rational(x, y):
    nx, dx = x.numer, x.denom
    ny, dy = y.numer, y.denom
    return Rational(dx * dy + nx * dy, dx + dy)

def mul_rational(x, y):
    return Rational(x.numer * y.numer, x.denom * y.denom)
Complex Numbers: the Rectangular Representation

class ComplexRI:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

@property
def magnitude(self):
    return (self.real ** 2 + self.imag ** 2) ** 0.5

@property
def angle(self):
    return atan2(self.imag, self.real)

def add_complex(z1, z2):
    return ComplexRI(z1.real + z2.real, z1.imag + z2.imag)

Might be either ComplexMA or ComplexRI instances

Special Methods

Special Methods for Arithmetic

Adding instances of user-defined classes with __add__.

class Rational:
...
def __add__(self, other):
    return add_rational(self, other)

Rational(1, 3) + Rational(1, 6)
Rational(1, 2)

We can also __add__ complex numbers, even with multiple representations. [Demo]

http://docs.python.org/py3k/reference/datamodel.html#special-method-names

The Independence of Data Types

Data abstraction and class definitions keep types separate

Some operations need to cross type boundaries

How do we add a complex number and a rational number together?

add_rational  mul_rational
Rational numbers as numerators & denominators

add_complex  mul_complex
Complex numbers as two-dimensional vectors

There are many different techniques for doing this!

Type Dispatching

Define a different function for each possible combination of types for which an operation (e.g., addition) is valid.

def complex(z):
    return type(z) in (ComplexRI, ComplexMA)
def rational(z):
    return type(z) is Rational
def add_complex_and_rational(z1, z2):
    if complex(z1) and complex(z2):
        return add_complex(z1, z2)
    elif complex(z1) and rational(z2):
        return add_complex_and_rational(z1, z2)
    elif rational(z1) and complex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        add_rational(z1, z2)

Converted to a real number (float)
Tag-Based Type Dispatching

Idea: Use a dictionary to dispatch on pairs of types.

```python
def type_tag(x):
    return type_tags[type(x)]

type_tags = {ComplexRI: 'com',
             ComplexMA: 'com',
             Rational: 'rat'}

def add(x1, x2):
    types = (type_tag(x1), type_tag(x2))
    return add_implementations[types](x1, x2)
```

(Demo)

Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary.

Extensible: Any new numeric type can "install" itself into the existing system by adding new entries to various dictionaries.

```python
def add(x1, x2):
    types = (type_tag(x1), type_tag(x2))
    return add_implementations[types](x1, x2)
```

Question 1: How many cross-type implementations are required for \( m \) types and \( n \) operations?

\[
m \cdot (m - 1) \cdot n
\]

Answer: http://goo.gl/FZKvgm

Type Dispatching Analysis

Data-Directed Programming

There's nothing addition-specific about `add`.

Idea: One function for all (operator, types) pairs

```python
def apply(operator_name, x, y):
    tags = (type_tag(x), type_tag(y))
    key = (operator_name, tags)
    return apply_implementations[key](x, y)
```

(Demo)
Type Coercion

Idea: Some types can be converted into other types
Takes advantage of structure in the type system

```python
def rational_to_complex(x):
    return Complex(x.numer / x.denom, 0)
```

```python
coercions = {('rat', 'com'): rational_to_complex}
```

**Question:** Can any numeric type be coerced into any other?

**Response:** http://goo.gl/FZKvgm

**Question:** Have we been repeating ourselves with data-directed programming?

---

Applying Operators with Coercion

1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```python
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
            tx, x = ty, coercions[(tx, ty)](x)
        elif (ty, tx) in coercions:
            ty, y = tx, coercions[(ty, tx)](y)
        else:
            return 'No coercion possible.'
    key = (operator_name, tx)
    return coerce_apply_implementations[key](x, y)
```

---

Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary.
Requires that all types can be coerced into a common type.
More sharing: All operators use the same coercion scheme.

<table>
<thead>
<tr>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Coerce</th>
<th>Type</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>