Cross-Type Arithmetic Examples

Currently, we can add rationals to rationals, but not rationals to complex numbers.

**Shared Interface**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational(1, 14) + Rational(2, 7)</td>
<td>3/7</td>
</tr>
<tr>
<td>Complex(1, 1) + Rational(0, 2, pi)</td>
<td>1+1</td>
</tr>
<tr>
<td>Rational(1, 14) * Rational(2, 7)</td>
<td>2/7</td>
</tr>
<tr>
<td>Complex(1, 1) * Rational(0, 2, pi)</td>
<td>1+1</td>
</tr>
</tbody>
</table>

**Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational(1, 14) + Complex(1, 0, pi)</td>
<td>1+1+1</td>
</tr>
<tr>
<td>Rational(1, 14) * Complex(1, 0, pi)</td>
<td>1+1+1</td>
</tr>
</tbody>
</table>

**Cross-type arithmetic**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex(1, 1) + Complex(1, 1, 0, pi)</td>
<td>1+1+1</td>
</tr>
<tr>
<td>Complex(1, 1) * Complex(1, 1, 0, pi)</td>
<td>1+1+1</td>
</tr>
</tbody>
</table>

Rational Numbers

```python
class Rational:
    '''A rational number represented as a numerator and denominator.'''
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numer = numer // g
        self.denom = denom // g
    def __repr__(self):
        return "Rational({0}, {1})".format(self.numer, self.denom)
    def add(self, other):
        nx, dy = other.numer, other.denom
        return Rational(nx*dy + ny*dx, dx*dy)
    def mul(self, other):
        numer = self.numer * other.numer
        denom = self.denom * other.denom
        return Rational(numer, denom)
```

Complex Numbers

```python
class Complex:
    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)
    @property
    def angleRadians(self):
        return self.magnitude * cos(self.angle)
```

More Generic Functions

A function might want to operate on multiple data types

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

**This lecture:**
- An arithmetic system over related types
- Operator overloading
- Type dispatching
- Type coercion

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

Special Method Names

**Common method names**

<table>
<thead>
<tr>
<th>Method</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>add</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>sub</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>mul</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>truediv</strong></td>
<td></td>
</tr>
<tr>
<td><strong>floordiv</strong></td>
<td></td>
</tr>
<tr>
<td><strong>mod</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>pow</strong>()</td>
<td></td>
</tr>
</tbody>
</table>

**Special method names**

<table>
<thead>
<tr>
<th>Method</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>repr</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>str</strong>()</td>
<td></td>
</tr>
<tr>
<td><strong>format</strong>()</td>
<td></td>
</tr>
</tbody>
</table>

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
- Homework 6 due Monday 10/28 @ 11:59pm (mail)
- Project 3 due Thursday 10/23 @ 11:59pm (BBG)
- Midterm 2 is on Monday 10/27
- Emphasis: mutable data, object-oriented programming, recursion, and recursive data
- Have an course conflict? Fill out the conflict form!
- Review session on Saturday 10/26

Homework 6 due Monday 10/20 @ 11:59pm (BIG!)

Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data

Project 3 due Thursday 10/23 @ 11:59pm (BIG!)

Special Method Names

**Common method names**

- __init__
- __add__
- __sub__
- __mul__
- __truediv__
- __floordiv__
- __mod__
- __pow__

**Special method names**

- __repr__
- __str__
- __format__

Cross-Type Arithmetic Examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational(1, 14) + Rational(2, 7)</td>
<td>3/7</td>
</tr>
<tr>
<td>Complex(1, 1) + Rational(0, 2, pi)</td>
<td>1+1</td>
</tr>
<tr>
<td>Rational(1, 14) * Rational(2, 7)</td>
<td>2/7</td>
</tr>
<tr>
<td>Complex(1, 1) * Rational(0, 2, pi)</td>
<td>1+1+1</td>
</tr>
</tbody>
</table>

Generic Functions of Multiple Arguments

Rational Numbers

```python
class Rational:
    '''A rational number represented as a numerator and denominator.'''
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numer = numer // g
        self.denom = denom // g
    def __repr__(self):
        return "Rational({0}, {1})".format(self.numer, self.denom)
    def add(self, other):
        nx, dy = other.numer, other.denom
        return Rational(nx*dy + ny*dx, dx*dy)
    def mul(self, other):
        numer = self.numer * other.numer
        denom = self.denom * other.denom
        return Rational(numer, denom)
```

Complex Numbers

```python
class Complex:
    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)
    @property
    def angleRadians(self):
        return self.magnitude * cos(self.angle)
```

More Generic Functions

A function might want to operate on multiple data types

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

**This lecture:**
- An arithmetic system over related types
- Operator overloading
- Type dispatching
- Type coercion

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

Certain names are special because they have built-in behavior:

- `__init__`: Method invoked automatically when an object is constructed
- `__repr__`: Method invoked to display an object as a string
- `__add__`: Method invoked to add one object to another
- `__bool__`: Method invoked to convert an object to True or False

These names always start and end with two underscores.

```python
>>> m = 4
>>> n = 2
>>> m * n
8
```

**Type Dispatching**

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

```python
class Number:
    def __add__(self, other):
        return self + other

class Rational(Number):
    def __add__(self, other):
        if isinstance(other, Number):
            return self + other
        else:
            return self + Rational(other)
```

**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 the cross-type function dictionaries.

**The Independence of Data Types**

Data abstraction and class definitions keep types separate:

Some operations need access to the implementation of two different abstractions.

```python
def add_complex_and_rational(c, r):
    return Complex(c.real + r.numer/c.denom, c.imag + r.numer/r.denom)
```

**Special Methods**

Adding instances of user-defined classes invokes the `__add__` method:

```python
class Number:
    def __add__(self, other):
        return self + other

class Rational(Number):
    def __add__(self, other):
        return self + Rational(other)
```

We can also `__add__` complex numbers, even with multiple representations:

```python
>>> add_complex_and_rational(Rational(1, 3), 1)
Complex(1.3333333333333333, 0)
```

**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 the cross-type function dictionaries.
Type Coercion

Coercion

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

```python
def rational_to_complex(r):
    """Return complex equal to rational."""
    return Complex(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?
Question: Can any two numeric types be coerced into a common type?
Question: Is coercion exact?

Applying Operators with Coercion

```python
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other
        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)
        elif (other.type_tag, self.type_tag) in self.coercions:
            return (self, other.coerce_to(self.type_tag))
    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

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

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>rational_to_complex</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>