61A Lecture 19

Wednesday, October 15

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
- Homework 6 due Monday 10/20 @ 11:59pm (small)
- Project 3 due Thursday 10/23 @ 11:59pm (BIG!)
- Midterm 2 is on Monday 10/27 7pm–9pm
  - 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 3pm–4:30pm and 4:30pm–6pm in 2050 VLSB

Generic Functions of Multiple Arguments

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.

Rational Numbers

```
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, dx = self.numer, self.denom
        ny, 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)
```

Greatest common divisor

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

(Demo)

Complex Numbers

```
class Complex:
    def add(self, other):
        return ComplexRI(self.real + other.real,
                         self.imag + other.imag)
    def mul(self, other):
        return ComplexMA(self.magnitude * other.magnitude,
                         self.angle + other.angle)
```

```
class ComplexRI(Complex):
    """A rectangular representation."""
    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)
```

```
class ComplexMA(Complex):
    """A polar representation."""
    def __init__(self, magnitude, angle):
        self.magnitude = magnitude
        self.angle = angle

    @property
    def real(self):
        return self.magnitude * cos(self.angle)

    @property
    def imag(self):
        return self.magnitude * sin(self.angle)
```

(Demo)

Cross-Type Arithmetic Examples

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

Shared interface

```
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```

$\frac{3}{14} + \frac{2}{7}$

$i \cdot i$

Operators

```
>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)
>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)
```

$\frac{3}{14} + \frac{2}{7}$

$i \cdot i$

Cross-type arithmetic

```
>>> Rational(1, 2) + ComplexRI(0.5, 2)
ComplexRI(1, 2)
>>> ComplexMA(2, 0.5 * pi) * Rational(3, 2)
ComplexMA(3, 0.5 * pi)
```

$\frac{1}{2} + (0.5 + 2 \cdot i)$

$2 \cdot i \cdot \frac{3}{2}$

Special Method Names

Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

`__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

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```

Same behavior using methods

```
>>> zero, one, two = 0, 1, 2
>>> one.__add__(two)
3
>>> zero.__bool__(), one.__bool__()
(False, True)
```


Special Methods

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

```
class Number:
    """A number."""
    def __add__(self, other):
        return self.add(other)

    def __mul__(self, other):
        return self.mul(other)

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

```
class Rational(Number):
    def add(self, other):
        ...
    def mul(self, other):
        ...

class Complex(Number):
    def add(self, other):
        ...
    def mul(self, other):
        ...
```

We can also `__add__` complex numbers, even with multiple representations (Demo)

http://getpython3.com/diveintopython3/special-method-names.html

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

Type Dispatching

The Independence of Data Types

Data abstraction and class definitions keep types separate

Some operations need access to the implementation of two different abstractions

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

Rational numbers as numerators & denominators **&** Complex numbers as two-dimensional vectors

```
def add_complex_and_rational(c, r):
    """Return c + r for complex c and rational r."""
    return ComplexRI(c.real + r.numer/r.denom, c.imag)
```


Type Dispatching

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

```
Rational.type_tag = "rat"
Complex.type_tag = "com"
```

Same tag: same interface

```
class Number:
    def __add__(self, other):
        if self.type_tag == other.type_tag:
            return self.add(other)
        elif (self.type_tag, other.type_tag) in self.adders:
            return self.cross_apply(other, self.adders)

    def cross_apply(self, other, cross_fns):
        cross_fn = cross_fns[(self.type_tag, other.type_tag)]
        return cross_fn(self, other)

    adders = {("com", "rat"): add_complex_and_rational,
              ("rat", "com"): add_rational_and_complex}
```

Defer to add method

All forms of cross-type addition for self

(Demo)

Type Dispatching Analysis

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

```
Number.adders[(tag0, tag1)] = add_tag0_and_tag1
```

Question: How many **cross-type implementations** are required for *m* types and *n* operations?

m n $m \cdot n$ $m^2 \cdot n$ $m^2 \cdot n^2$

$$m \cdot (m-1) \cdot n$$

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

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

Type Coercion

Coercion

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```
def rational_to_complex(r):
    """Return complex equal to rational."""
    return ComplexRI(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

```
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}
```

Always defer to add method

Same interface: no change required

(Demo)

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

Arg 1	Arg 2	Add	Multiply
Complex	Complex		
Rational	Rational		
Complex	Rational		
Rational	Complex		

From	To	Coerce
Complex	Rational	
Rational	Complex	

Type	Add	Multiply
Complex		
Rational		