

## 61A Lecture 19

Wednesday, October 10

### Generic Functions, Continued

A function might want to operate on multiple data types

Last time:

- Polymorphic functions using message passing
- Interfaces: collections of messages with a meaning for each
- Two interchangeable implementations of complex numbers

Today:

- An arithmetic system over related types
- Type dispatching instead of message passing
- 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(object):  
    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(nx * dy + ny * dx, dx * dy)  
  
    def mul_rational(x, y):  
        return Rational(x.numer * y.numer, x.denom * y.denom)
```

### Complex Numbers: the Rectangular Representation

```
class ComplexRI(object):  
    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 __repr__(self):  
        return 'ComplexRI({0}, {1})'.format(self.real,  
                                            self.imag)  
  
    def add_complex(z1, z2): → Might be either ComplexMA or ComplexRI instances  
        return ComplexRI(z1.real + z2.real,  
                         z1.imag + z2.imag)
```

### Special Methods

Adding instances of user-defined classes with `__add__`.

Demo

```
>>> ComplexRI(1, 2) + ComplexMA(2, 0)  
ComplexRI(3.0, 2.0)  
>>> ComplexRI(0, 1) * ComplexRI(0, 1)  
ComplexMA(1.0, 3.141592653589793)
```

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

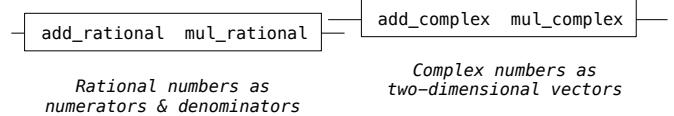
<http://docs.python.org/py3k/reference/datamodel.html#special-method-names>

### 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?



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 iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)

def isrational(z):
    return type(z) is Rational
    Converted to a real number (float)

def add_complex_and_rational(z, r):
    return ComplexRI(z.real + r.numer/r.denom, z.imag)

def add_by_type_dispatching(z1, z2):
    """Add z1 and z2, which may be complex or rational."""
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        add_rational(z1, z2)
```

Demo

7

## Tag-Based Type Dispatching

**Idea:** Use dictionaries to dispatch on type

```
def type_tag(x):
    return type_tag.tags[type(x)]

type_tag.tags = {ComplexRI: 'com',
                ComplexMA: 'com',
                Rational: 'rat'}
    Declares that ComplexRI and ComplexMA should be treated uniformly

def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add.implementations[types](z1, z2)

add.implementations = {}
add.implementations[('com', 'com')] = add_complex
add.implementations[('rat', 'rat')] = add_rational
add.implementations[('com', 'rat')] = add_complex_and_rational
add.implementations[('rat', 'com')] = add_rational_and_complex
    add_rational_and_complex

lambda r, z: add_complex_and_rational(z, r)
```

8

## Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary, but use abstract data types

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

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add.implementations[types](z1, z2)
```

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

integer, rational, real, complex       $m \cdot (m - 1) \cdot n$       add, subtract, multiply, divide  
 $4 \cdot (4 - 1) \cdot 4 = 48$

9

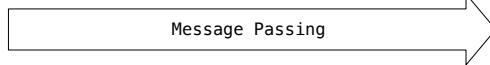


## Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary, but use abstract data types

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

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



10

## Data-Directed Programming

There's nothing addition-specific about add\_by\_type

**Idea:** One dispatch function for (operator, types) pairs

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

Demo

11

## Coercion

**Idea:** Some types can be converted into other types

Takes advantage of structure in the type system

```
>>> def rational_to_complex(x):
            return ComplexRI(x.numer/x.denom, 0)

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

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

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

12

## Applying Operators with Coercion

1. Attempt to coerce arguments into values of the same type

2. Apply type-specific (not cross-type) operations

```
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.'
    assert tx == ty
    key = (operator_name, tx)
    return coerce_apply.implementations[key](x, y)
```

Demo

## Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary, but use abstract data types

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		

13

14