61A Lecture 21

Wednesday, October 23

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 5:30-7 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
  - 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

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.*

Representing Numbers

Rational Numbers

Rational numbers represented as a numerator and denominator

```
class Rational:

    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_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)
```

Greatest common divisor

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 __repr__(self):
        return 'ComplexRI({0}, {1})'.format(self.real,
                                            self.imag)
```

Might be either ComplexMA or ComplexRI instances

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


Special Methods for Arithmetic

Special Methods

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

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

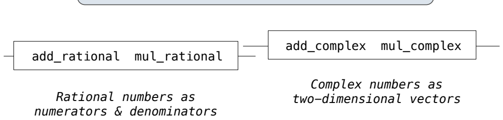
Type Dispatching

The Independence of Data Types

Data abstraction and class definitions keep types separate

Some operations need 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 complex(z):
    return type(z) in (ComplexRI, ComplexMA)

def rational(z):
    return type(z) is Rational

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 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.

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

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

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

Declares that ComplexRI and ComplexMA should be treated the same

(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 various dictionaries

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

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

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

Respond: http://goo.gl/FZKvgm

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

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

Data-Directed Programming

Data-Directed Programming

There's nothing addition-specific about add.

Idea: One function for all (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)

Type Coercion

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?

Respond: 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

```
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.

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		