Announcements

☐ HW7 due tonight

☐ Ants project due Monday

☐ HW8 due next Wednesday at 7pm

☐ Midterm 2 next Thursday at 7pm
Interfaces

Message passing allows **different data types** to respond to the same message.

A shared message that elicits similar behavior from different object classes is a powerful method of abstraction.

An *interface* is a set of shared messages, along with a specification of what they mean.

In languages like Python and Ruby, interfaces are implicitly implemented by providing the right methods with the correct behavior

- *If it quacks like a duck...*

Other languages require interfaces to be explicitly implemented
Example: Rational Numbers

class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator, self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator, self.denominator)
    def __add__(self, num):
        return add_rational(self, num)
    def __mul__(self, num):
        return mul_rational(self, num)
    def __eq__(self, num):
        return eq_rational(self, num)
Often, we want the value of instance attributes to be linked.

```python
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
>>> f.float_value
0.8
>>> f.denominator -= 3
>>> f.float_value
2.0
```

The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance.

It allows zero-argument methods to be called without an explicit call expression.
Multiple Representations of Abstract Data

Rectangular and polar representations for complex numbers

Most operations don't care about the representation.

Some mathematical operations are easier on one than the other.
Complex numbers as whole data values
- add_complex
- mul_complex

Complex numbers as two-dimensional vectors
- real
- imag
- magnitude
- angle

Rectangular representation
Polar representation
An Interface for Complex Numbers

All complex numbers should have real and imag components.

All complex numbers should have a magnitude and angle.

Using this interface, we can implement complex arithmetic:

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

def mul_complex(z1, z2):
    return ComplexMA(z1.magnitude * z2.magnitude,
                     z1.angle + z2.angle)
```
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)
class ComplexMA(object):
    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)

    def __repr__(self):
        return 'ComplexMA({0}, {1})'.format(self.magnitude, self.angle)
Using Complex Numbers

Either type of complex number can be passed as either argument to `add_complex` or `mul_complex`:

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

def mul_complex(z1, z2):
    return ComplexMA(z1.magnitude * z2.magnitude,
                     z1.angle + z2.angle)
```

```python
>>> from math import pi
>>> add_complex(ComplexRI(1, 2), ComplexMA(2, pi/2))
ComplexRI(1.0000000000000002, 4.0)
>>> mul_complex(ComplexRI(0, 1), ComplexRI(0, 1))
ComplexMA(1.0, 3.141592653589793)
```

We can also define `__add__` and `__mul__` in both classes.
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?

Rational numbers as numerators & denominators

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

```python
def iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)
def isrational(z):
    return type(z) is Rational
def add_complex_and_rational(z, r):
    return ComplexRI(z.real + r.numerator / r.denominator, 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)
```

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

Idea: Use dictionaries to dispatch on type (like we did for message passing)

```python
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)

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

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

Declares that `ComplexRI` and `ComplexMA` should be treated uniformly
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

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

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

- integer, rational, real, complex
- add, subtract, multiply, divide

$m \cdot (m - 1) \cdot n = \boxed{4 \cdot (4 - 1) \cdot 4 = 48}$
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.

<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>

Message Passing

Type Dispatching