61A Lecture 17

Wednesday, March 4
Announcements

• Delayed: Hog contest winners will be announced Friday 3/6 in lecture
Announcements

• Delayed: Hog contest winners will be announced Friday 3/6 in lecture

• Quiz 2 due Thursday 3/5 @ 11:59pm (challenging!)
Announcements

• Delayed: Hog contest winners will be announced Friday 3/6 in lecture

• Quiz 2 due Thursday 3/5 @ 11:59pm (challenging!)

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)
Announcements

• Delayed: Hog contest winners will be announced Friday 3/6 in lecture

• Quiz 2 due Thursday 3/5 @ 11:59pm (challenging!)

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)

• Delayed: Homework 6 due Monday 3/16 @ 11:59pm
Announcements

• Delayed: Hog contest winners will be announced Friday 3/6 in lecture

• Quiz 2 due Thursday 3/5 @ 11:59pm (challenging!)

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)

• Delayed: Homework 6 due Monday 3/16 @ 11:59pm

• Midterm 2 is on Thursday 3/19 7pm–9pm
Announcements

• Delayed: Hog contest winners will be announced Friday 3/6 in lecture

• Quiz 2 due Thursday 3/5 @ 11:59pm (challenging!)

• Project 3 due Thursday 3/12 @ 11:59pm (get started now!)

• Delayed: Homework 6 due Monday 3/16 @ 11:59pm

• Midterm 2 is on Thursday 3/19 7pm–9pm

  † Emphasis: mutable data, object-oriented programming, recursion, and recursive data
Generic Functions of Multiple Arguments
More Generic Functions
More Generic Functions

A function might want to operate on multiple data types
More Generic Functions

A function might want to operate on multiple data types

Last lecture:
More Generic Functions

A function might want to operate on multiple data types

Last lecture:
- Polymorphic functions using shared messages
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
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
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:
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
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
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
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
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)
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)

Greatest common divisor
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)
class Rational:
    """A rational number represented as a numerator and denominator."""
    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(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)
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)
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)

(Demo)
Complex Numbers
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)
Complex Numbers

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)

class ComplexRI(Complex):
    """A rectangular representation."""
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

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)
Complex Numbers

class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    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)
Complex Numbers

class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    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
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

\[
\frac{3}{14} + \frac{2}{7}
\]
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```
Cross-Type Arithmetic Examples

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

```
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

```
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```

```
3/14 + 2/7
```

```
i * i
```
Cross-Type Arithmetic Examples

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

\[
\frac{3}{14} + \frac{2}{7}
\]

\[
i \cdot i
\]

Shared interface

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)

>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)

>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```

```python
>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```

```python
>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)
```

```python
>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)
```

```python
>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)

>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)

>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)

>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)
```

\[
\begin{align*}
\text{Shared interface} & \\
\text{Operators} & \\
& \text{Rational}(3, 14) + \text{Rational}(2, 7) = \frac{3}{14} + \frac{2}{7} \quad i \cdot i \\
& \text{ComplexRI}(0, 1) \cdot \text{ComplexMA}(1, 0.5 \times \pi) = i \cdot i
\end{align*}
\]
Cross-Type Arithmetic Examples

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

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

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

```
>>> Rational(1, 2) + ComplexRI(0.5, 2)
ComplexRI(1, 2)
```
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)

3 \cdot 7 + 2 \cdot 14
14

3 + 2
7

i \cdot i

>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)

i \cdot i

>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)

3 \cdot 7 + 2 \cdot 14
14

3 + 2
7

>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)

i \cdot i

>>> Rational(1, 2) + ComplexRI(0.5, 2)
ComplexRI(1, 2)

1 + (0.5 + 2 \cdot i)
2

1 + 0.5 + 2 \cdot i
2
```

Shared interface

Operators
Cross-Type Arithmetic Examples

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

```python
Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)

ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)

Rational(3, 14) + Rational(2, 7)
Rational(1, 2)

ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)

Rational(1, 2) + ComplexRI(0.5, 2)
ComplexRI(1, 2)

ComplexMA(2, 0.5 * pi) * Rational(3, 2)
ComplexMA(3, 0.5 * pi)
```

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

$$i \cdot i$$

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

$$i \cdot i$$

$$\frac{1}{2} + (0.5 + 2 \cdot i)$$
Cross-Type Arithmetic Examples

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

```python
>>> Rational(3, 14).add(Rational(2, 7))
Rational(1, 2)

Rational(1, 2)

>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))
ComplexMA(1, 1 * pi)

ComplexMA(1, 1 * pi)

>>> Rational(3, 14) + Rational(2, 7)
Rational(1, 2)

Rational(1, 2)

>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)
ComplexMA(1, 1 * pi)

ComplexMA(1, 1 * pi)

>>> Rational(1, 2) + ComplexRI(0.5, 2)
ComplexRI(1, 2)

ComplexRI(1, 2)

>>> ComplexMA(2, 0.5 * pi) * Rational(3, 2)
ComplexMA(3, 0.5 * pi)

ComplexMA(3, 0.5 * pi)
```
Cross-Type Arithmetic Examples

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

<table>
<thead>
<tr>
<th>Shared interface</th>
<th>Operators</th>
<th>Cross-type arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt; Rational(3, 14).add(Rational(2, 7))</td>
<td>&gt;&gt;&gt; ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</td>
<td>&gt;&gt;&gt; ComplexMA(2, 0.5 * pi) * Rational(3, 2)</td>
</tr>
<tr>
<td>Rational(1, 2)</td>
<td>ComplexMA(1, 1 * pi)</td>
<td>ComplexMA(3, 0.5 * pi)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; Rational(3, 14) + Rational(2, 7)</td>
<td>&gt;&gt;&gt; ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)</td>
<td>&gt;&gt;&gt; ComplexMA(2, 0.5 * pi) * Rational(3, 2)</td>
</tr>
<tr>
<td>Rational(1, 2)</td>
<td>ComplexMA(1, 1 * pi)</td>
<td>ComplexMA(3, 0.5 * pi)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; 3/14 + 2/7</td>
<td>&gt;&gt;&gt; i·i</td>
<td>&gt;&gt;&gt; 1/2 + (0.5 + 2·i)</td>
</tr>
<tr>
<td>3/14 + 2/7</td>
<td>i·i</td>
<td>2·i·3/2</td>
</tr>
</tbody>
</table>
Special Method Names
Special Method Names in Python
Special Method Names in Python

Certain names are special because they have built-in behavior
Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores
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__
Special Method Names in Python

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

These names always start and end with two underscores.

```python
__init__  Method invoked automatically when an object is constructed
```
Special Method Names in Python

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

These names always start and end with two underscores:

```python
__init__  Method invoked automatically when an object is constructed
__repr__  
```
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
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__`
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
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 compare whether an object is truthy or falsy.
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
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
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

```python
>>> zero, one, two = 0, 1, 2
>>> one + two
3
```
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.

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

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

Same behavior using methods.
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.

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

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

```python
>>> zero, one, two = 0, 1, 2
>>> one.__add__(two)
3
```
Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

```python
__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
```

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

Same behavior using methods

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

Adding instances of user-defined classes invokes the \texttt{\_\_add\_\_} method
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)
**Special Methods**

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

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

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

Adding instances of user-defined classes invokes the __add__ method

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

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

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

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

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

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

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

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

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

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


http://docs.python.org/py3k/reference/datamodel.html#special-method-names
Type Dispatching
The Independence of Data Types
The Independence of Data Types

Data abstraction and class definitions keep types separate
The Independence of Data Types

Data abstraction and class definitions keep types separate.

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

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

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

\[
\text{Rational.type_tag = "rat"} \\
\text{Complex.type_tag = "com"}
\]

Same tag: same interface

```python
class Number:
    def __add__(self, other):
        if self.type_tag == other.type_tag:
            return self.add(other)
```

Type Dispatching

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

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

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

Same tag: same interface
Defer to add method
Type Dispatching

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

```python
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)
```

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

Same tag: same interface

Defer to add method
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"

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)
        else:
            # Implement additional methods for cross-type addition

Same tag: same interface
Defer to add method
All forms of cross-type addition for self
Type Dispatching

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

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

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)

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

```python
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}
```
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"
```

```
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)
        else:
            raise TypeError("Incompatible types")

    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}
```

(Demo)
Type Dispatching Analysis
Type Dispatching Analysis
Type Dispatching Analysis

Minimal violation of abstraction barriers: we define cross-type functions as necessary
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 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

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

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

\[
\text{Number.adders}[(\text{tag0, tag1})] = \text{add}_\text{tag0\_and\_tag1}
\]

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

\[
m \quad n \quad m \cdot n \quad m^2 \cdot n \quad m^2 \cdot n^2
\]

\[
m \cdot (m - 1) \cdot n
\]
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.
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

<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>
Type Coercion
Coercion
Coercion

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

Idea: Some types can be converted into other types

Takes advantage of structure in the type system
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 ComplexRI(r.numer/r.denom, 0)
```
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 ComplexRI(r.numer/r.denom, 0)
```

Question: Can any numeric type be coerced into any other?
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 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?
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 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
Applying Operators with Coercion

class Number:
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)
Applying Operators with Coercion

class Number:
def add(self, other):
    x, y = self.coerce(other)
    return x.add(y)

Always defer to add method
Applying Operators with Coercion

class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        Always defer to add method
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

Always defer to add method
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

Always defer to add method

Same interface: no change required
```
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:
            # Always defer to add method
            pass
        else:
            # Always defer to add method
            pass

    # Same interface: no change required
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:
            coercions = {('rat', 'com'): rational_to_complex}

        Always defer to add method

        Same interface: no change required
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)

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

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

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

- Always defer to `add` method
- Same interface: no change required
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, other.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}
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}
Coercion Analysis
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary
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
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
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>
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>

From | To | Coerce
---|----|--
Complex| Rational| 
Rational| 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>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>