61A Lecture 17

Wednesday, March 4

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

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

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

• 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: 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

• 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

• 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

A function might want to operate on multiple data types

A function might want to operate on multiple data types

Last lecture:

A function might want to operate on multiple data types

Last lecture:

• Polymorphic functions using shared messages

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

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

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

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

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

- An arithmetic system over related types
- Operator overloading

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

- An arithmetic system over related types
- Operator overloading
- •Type dispatching

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

- An arithmetic system over related types
- Operator overloading
- •Type dispatching
- •Type coercion

A function might want to operate on multiple data types

Last lecture:

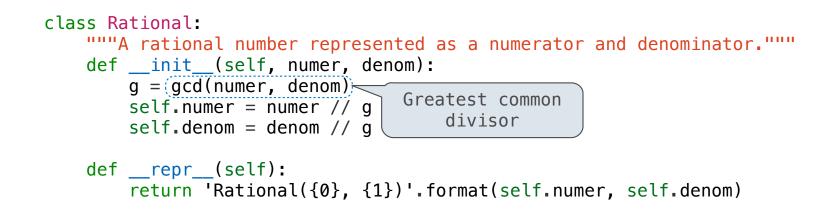
- Polymorphic functions using shared messages
- Interfaces: collections of messages that have specific behavior conditions
- Two interchangeable implementations of complex numbers

This lecture:

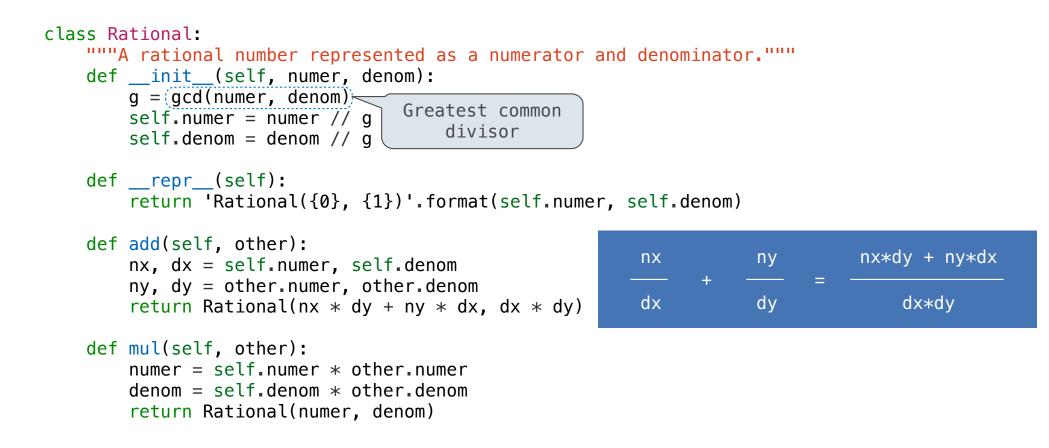
- An arithmetic system over related types
- Operator overloading
- •Type dispatching
- •Type coercion

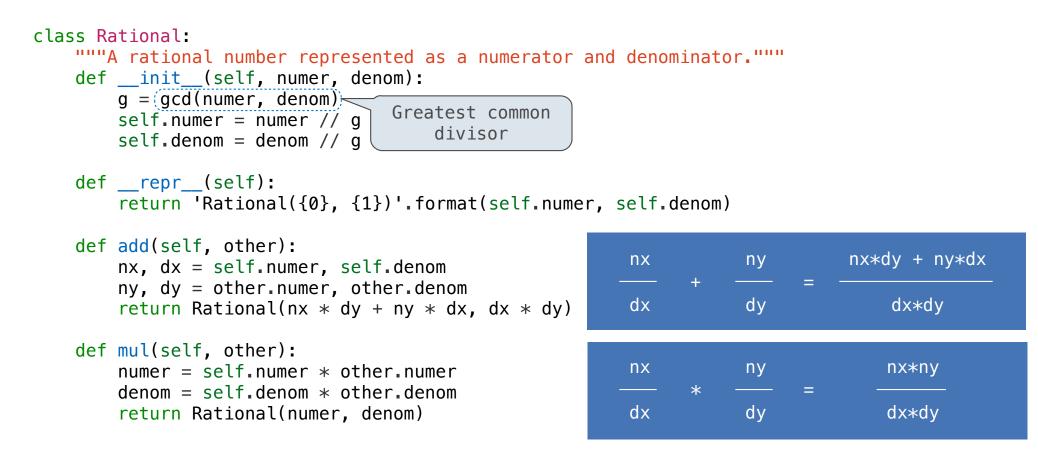
```
class Rational:
    """A rational number represented as a numerator and denominator."""
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numer = numer // g
        self.denom = denom // g

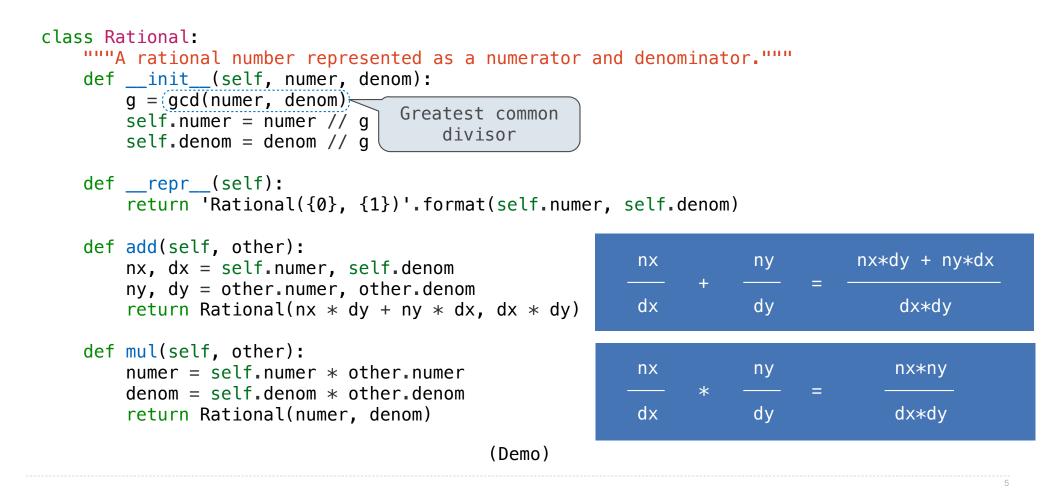
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numer, self.denom)
```



```
class Rational:
    """A rational number represented as a numerator and denominator."""
    def init (self, numer, denom):
        g = (gcd(numer, denom))
                                  Greatest common
        self.numer = numer // g
                                      divisor
        self.denom = denom // q
    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 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 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 ComplexMA(Complex):
class ComplexRI(Complex):
   """A rectangular representation."""
                                                             """A polar representation."""
   def init (self, real, imag):
                                                             def __init__(self, magnitude, angle):
       self.real = real
                                                                 self.magnitude = magnitude
       self.imag = imag
                                                                 self.angle = angle
   @property
                                                             @property
   def magnitude(self):
                                                             def real(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5
                                                                 return self.magnitude * cos(self.angle)
   @property
                                                             @property
   def angle(self):
                                                             def imag(self):
        return atan2(self.imag, self.real)
                                                                 return self.magnitude * sin(self.angle)
```

```
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 ComplexMA(Complex):
class ComplexRI(Complex):
                                                             """A polar representation."""
   """A rectangular representation."""
   def init (self, real, imag):
                                                             def __init__(self, magnitude, angle):
       self.real = real
                                                                 self.magnitude = magnitude
       self.imag = imag
                                                                 self.angle = angle
   @property
                                                             @property
   def magnitude(self):
                                                             def real(self):
       return (self.real ** 2 + self.imag ** 2) ** 0.5
                                                                 return self.magnitude * cos(self.angle)
   @property
                                                             @property
                                                             def imag(self):
   def angle(self):
       return atan2(self.imag, self.real)
                                                                 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

```
>>> 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

```
>>> Rational(3, 14).add(Rational(2, 7)) \frac{3}{14} + \frac{2}{7}
Rational(1, 2)
```

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

Rational(1, 2)

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

ComplexMA(1, 1 * pi)
```

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)) $i \cdot i$ ComplexMA(1, 1 * pi)

	<pre>>>> Rational(3, 14).add(Rational(2, 7))</pre>	$\frac{3}{14} + \frac{2}{7}$
Shared	Rational(1, 2)	14 7
interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i \cdot i$
	ComplexMA(1, 1 * pi)	

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

Shared interface	<pre>>>> Rational(3, 14).add(Rational(2, 7)) Rational(1, 2)</pre>	$\frac{3}{14} + \frac{2}{7}$
	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi)) ComplexMA(1, 1 * pi)</pre>	$i\cdot i$
	<pre>>>> Rational(3, 14) + Rational(2, 7) Rational(1, 2)</pre>	$\frac{3}{14} + \frac{2}{7}$

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

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) >>> Rational(3, 14) + Rational(2, 7) Rational(1, 2) >>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi) ComplexMA(1, 1 * pi) $i \cdot i$

	<pre>>>> Rational(3, 14).add(Rational(2, 7))</pre>	$\frac{3}{14} + \frac{2}{7}$
Shared	Rational(1, 2)	14 7
interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(3, 14) + Rational(2, 7)</pre>	$\frac{3}{3} + \frac{2}{3}$
•	Rational(1, 2)	14 7
Operators	>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)	$i\cdot i$
	ComplexMA(1, 1 * pi)	

	<pre>>>> Rational(3, 14).add(Rational(2, 7))</pre>	$\frac{3}{14} + \frac{2}{7}$
Shared	Rational(1, 2)	14 (
interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(3, 14) + Rational(2, 7)</pre>	$\frac{3}{14} + \frac{2}{7}$
0	Rational(1, 2)	14 7
Operators	>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(1, 2) + ComplexRI(0.5, 2)</pre>	
	ComplexRI(1, 2)	

	<pre>>>> Rational(3, 14).add(Rational(2, 7))</pre>	$\frac{3}{14} + \frac{2}{7}$
Shared	Rational(1, 2)	14 7
interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(3, 14) + Rational(2, 7)</pre>	$\frac{3}{14} + \frac{2}{7}$
0	Rational(1, 2)	14 (
Operators	>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(1, 2) + ComplexRI(0.5, 2)</pre>	$\frac{1}{2} + (0.5 + 2 \cdot i)$
	ComplexRI(1, 2)	2

Shared	<pre>>>> Rational(3, 14).add(Rational(2, 7)) Rational(1, 2)</pre>	$\frac{3}{14} + \frac{2}{7}$
interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(3, 14) + Rational(2, 7)</pre>	$\frac{3}{14} + \frac{2}{7}$
	Rational(1, 2)	14 (
Operators	>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(1, 2) + ComplexRI(0.5, 2)</pre>	$\frac{1}{2} + (0.5 + 2 \cdot i)$
	ComplexRI(1, 2)	2
	<pre>>>> ComplexMA(2, 0.5 * pi) * Rational(3, 2)</pre>	
	ComplexMA(3, 0.5 * pi)	

		<pre>>>> Rational(3, 14).add(Rational(2, 7))</pre>	$\frac{3}{14} + \frac{2}{7}$
Shared	Shared	Rational(1, 2)	14 (
	interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i\cdot i$
		ComplexMA(1, 1 * pi)	
		<pre>>>> Rational(3, 14) + Rational(2, 7)</pre>	$\frac{3}{14} + \frac{2}{7}$
	Operators	Rational(1, 2)	14 (
Uperato	Operators	<pre>>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)</pre>	$i\cdot i$
		ComplexMA(1, 1 * pi)	
		<pre>>>> Rational(1, 2) + ComplexRI(0.5, 2)</pre>	$\frac{1}{2} + (0.5 + 2 \cdot i)$
		ComplexRI(1, 2)	2
		<pre>>>> ComplexMA(2, 0.5 * pi) * Rational(3, 2)</pre>	$2 \cdot i \cdot \frac{3}{2}$
		ComplexMA(3, 0.5 * pi)	2

	<pre>>>> Rational(3, 14).add(Rational(2, 7))</pre>	$\frac{3}{14} + \frac{2}{7}$
Shared	Rational(1, 2)	14 (
interface	<pre>>>> ComplexRI(0, 1).mul(ComplexMA(1, 0.5 * pi))</pre>	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(3, 14) + Rational(2, 7)</pre>	$\frac{3}{14} + \frac{2}{7}$
	Rational(1, 2)	14 7
Operators	>>> ComplexRI(0, 1) * ComplexMA(1, 0.5 * pi)	$i\cdot i$
	ComplexMA(1, 1 * pi)	
	<pre>>>> Rational(1, 2) + ComplexRI(0.5, 2)</pre>	$\frac{1}{2} + (0.5 + 2 \cdot i)$
Cross-type	ComplexRI(1, 2)	2
arithmetic	>>> ComplexMA(2, 0.5 * pi) * Rational(3, 2)	$2 \cdot i \cdot \frac{3}{2}$
	ComplexMA(3, 0.5 * pi)	2

Special Method Names

Certain names are special because they have built-in behavior

Certain names are special because they have built-in behavior

Certain names are special because they have built-in behavior

These names always start and end with two underscores

__init__

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

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__

Certain names are special because they have built-in behavior

- ___init___ Method invoked automatically when an object is constructed
- ___repr___ Method invoked to display an object as a string

Certain names are special because they have built-in behavior

init	Method invoked automatically when an object is constructed
repr	Method invoked to display an object as a string
add	

Certain names are special because they have built-in behavior

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

Certain names are special because they have built-in behavior

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	

Certain names are special because they have built-in behavior

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

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

Certain names are special because they have built-in behavior

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
```

Certain names are special because they have built-in behavior

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)
```

Certain names are special because they have built-in behavior

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

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

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

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._bool_(), one._bool_()
(False, True)
```

9

Adding instances of user-defined classes invokes the __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)
```

```
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

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)
```

Special Methods

```
Adding instances of user-defined classes invokes the __add__ method
    class Number:
                                                class Rational(Number):
        """A number."""
                                                    def add(self, other):
        def __add__(self, other):
            return self.add(other)
                                                    def mul(self, other):
                                                        . . .
        def mul (self, other):
            return self.mul(other)
                                                class Complex(Number):
                                                    def add(self, other):
    >>> Rational(1, 3) + Rational(1, 6)
                                                    def mul(self, other):
    Rational(1, 2)
                                                        . . .
```

We can also <u>__add__</u> complex numbers, even with multiple representations (Demo)

Special Methods

```
Adding instances of user-defined classes invokes the __add__ method
    class Number:
                                                class Rational(Number):
        """A number."""
                                                    def add(self, other):
        def __add__(self, other):
            return self.add(other)
                                                    def mul(self, other):
                                                        . . .
        def mul (self, other):
            return self.mul(other)
                                                class Complex(Number):
                                                    def add(self, other):
    >>> Rational(1, 3) + Rational(1, 6)
                                                    def mul(self, other):
    Rational(1, 2)
                                                        . . .
```

We can also <u>__add__</u> 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

Data abstraction and class definitions keep types separate

Data abstraction and class definitions keep types separate

Some operations need access to the implementation of two different abstractions

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?

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

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

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

&

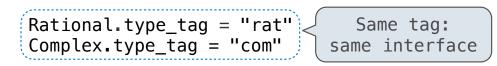
Complex numbers as two-dimensional vectors

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)

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"

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

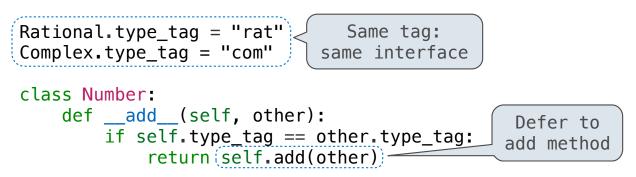


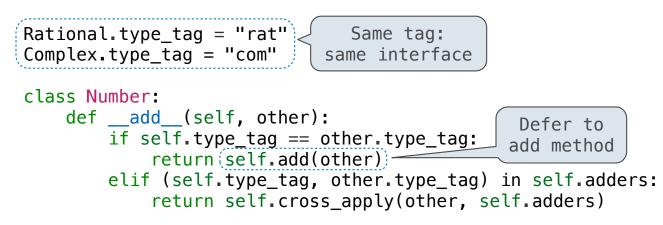
class Number:

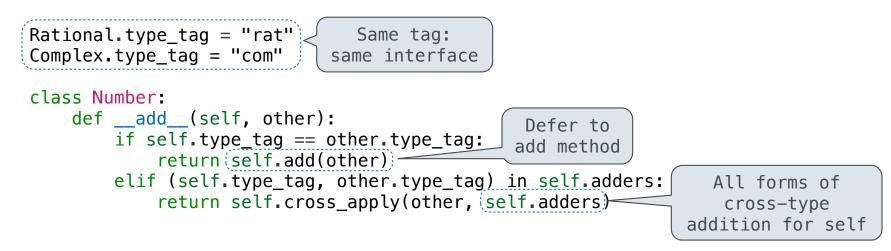
Define a different function for each possible combination of types for which an operation (e.g., addition) is valid

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

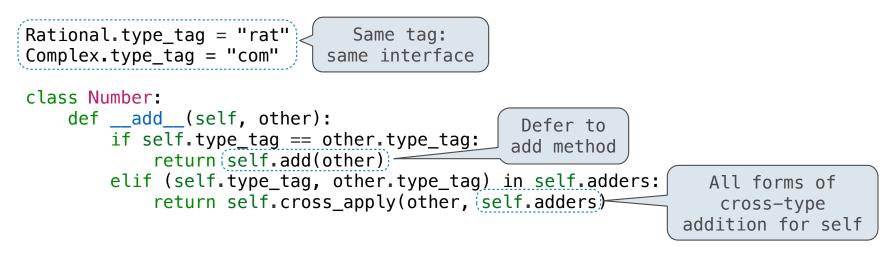
class Number: def __add__(self, other): if self.type_tag == other.type_tag: return self.add(other)

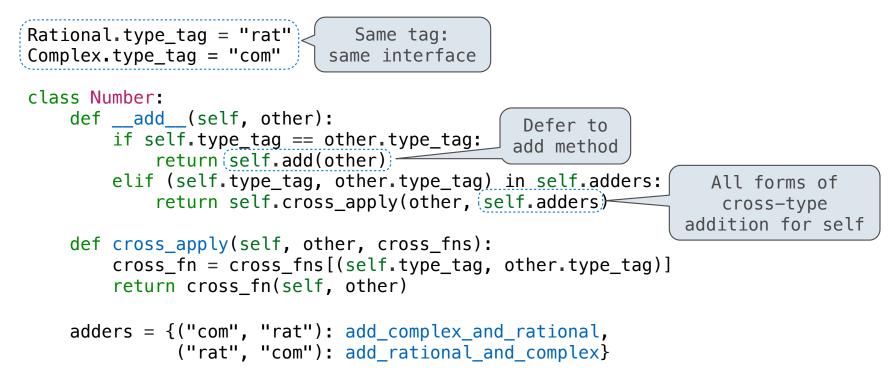


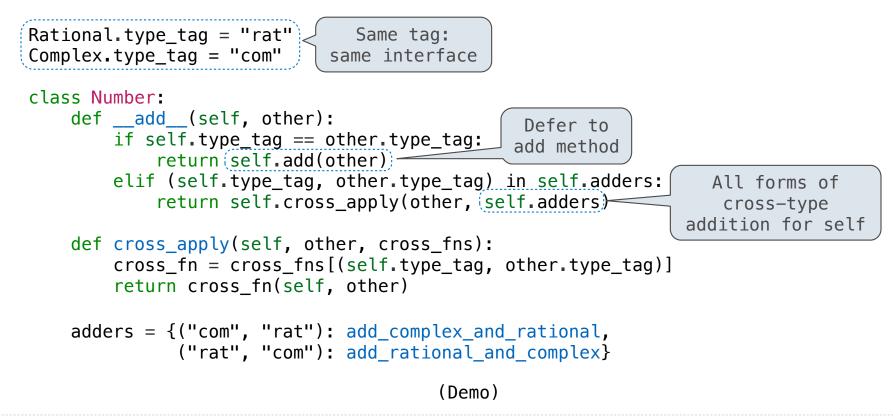




Define a different function for each possible combination of types for which an operation (e.g., addition) is valid







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

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

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

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?

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

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?

15

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

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

Coercion

Idea: Some types can be converted into other types

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

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)
```

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?

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?

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)
```

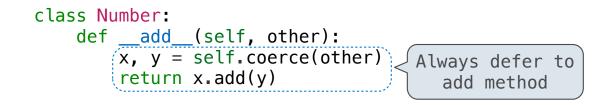
Question: Can any numeric type be coerced into any other?

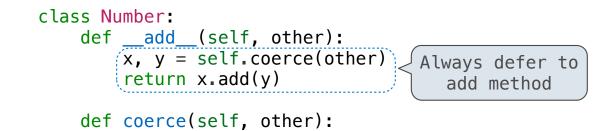
Question: Can any two numeric types be coerced into a common type?

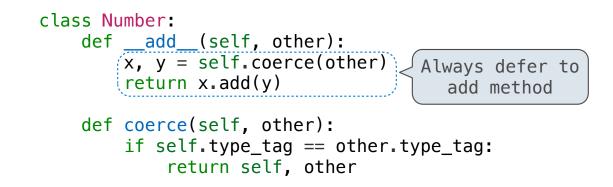
```
Question: Is coercion exact?
```

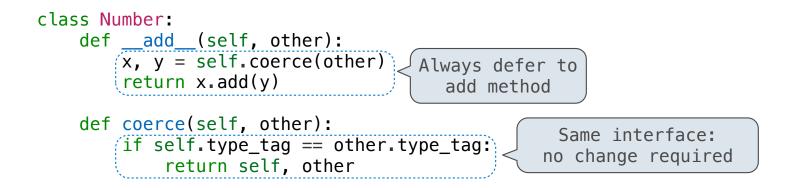
class Number:

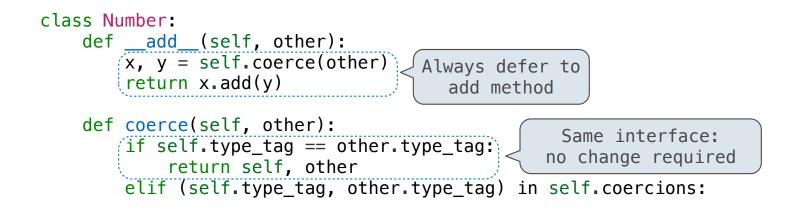
class Number: def __add__(self, other): x, y = self.coerce(other) return x.add(y)

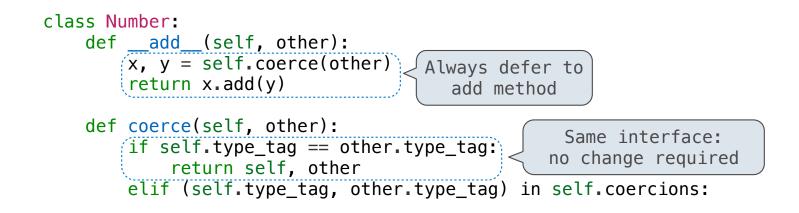




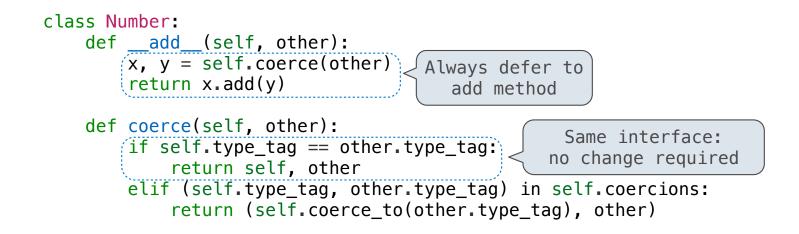




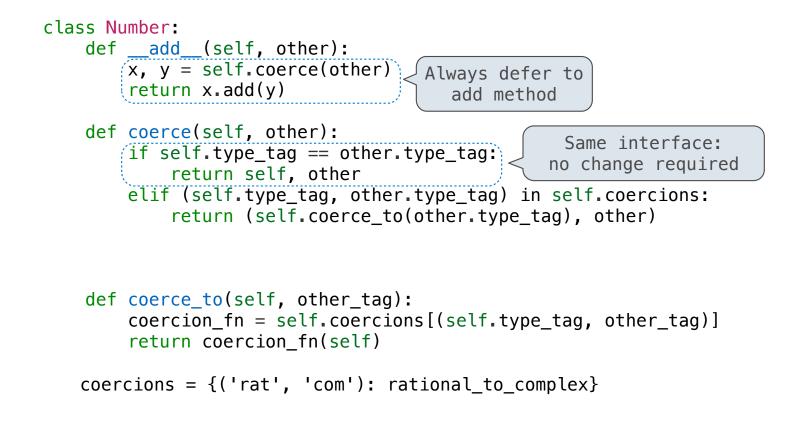


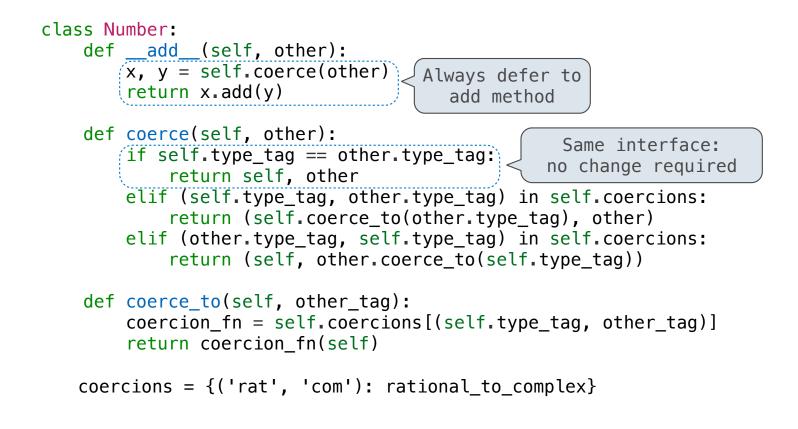


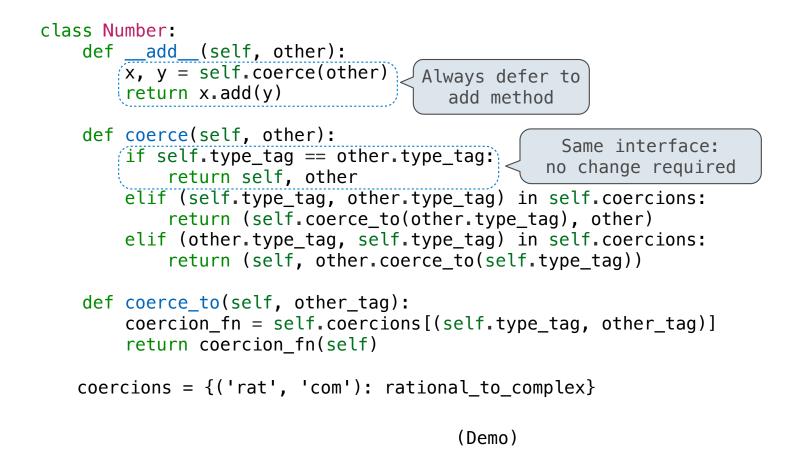
coercions = {('rat', 'com'): rational_to_complex}



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







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

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

Requires that all types can be coerced into a common type

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

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		

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		

		\square
From	То	Coerce
Complex	Rational	
Rational	Complex	

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	Arg 2		Add		Multiply	
Complex	Compl	Complex					
Rational	Ration	Rational					
Complex	Ration	al					
Rational Complex							
				\sum			
From	То	Coerce		Туре	Add	Multiply	
Complex	Rational			Complex			
Rational	Complex			Rational			