61A Lecture 8

Wednesday, September 12

Data Abstraction

• Compound objects combine primitive objects together

- Compound objects combine primitive objects together
- A date: a year, a month, and a day

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:
 - How data are represented (as parts)

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:
 - How data are represented (as parts)
 - How data are manipulated (as units)

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:
 - How data are represented (as parts)
 - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use

Data Abstraction

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:

How data are represented (as parts)

How data are manipulated (as units)

 Data abstraction: A methodology by which functions enforce an abstraction barrier between *representation* and *use* All Programmers Data Abstraction

- Compound objects combine primitive objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:

How data are represented (as parts)

How data are manipulated (as units)

 Data abstraction: A methodology by which functions enforce an abstraction barrier between *representation* and *use* All Programmers

> Great Programmers

numerator

denominator

numerator

denominator

Exact representation of fractions

numerator

denominator

Exact representation of fractions

A pair of integers

denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

• rational(n, d) returns a rational number x

denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

Constructor (rational(n, d) returns a rational number x

denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

Constructor (rational(n, d) returns a rational number x

• numer(x) returns the numerator of x

denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

Constructor (rational(n, d) returns a rational number x

- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

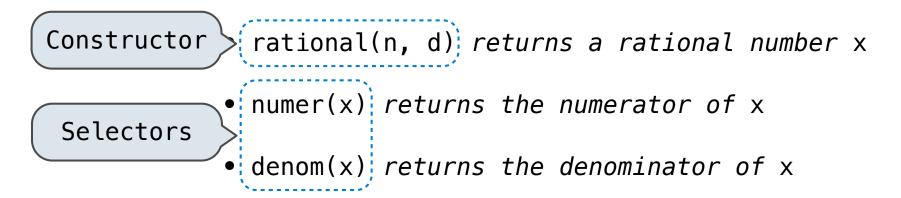
denominator

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:



Example:

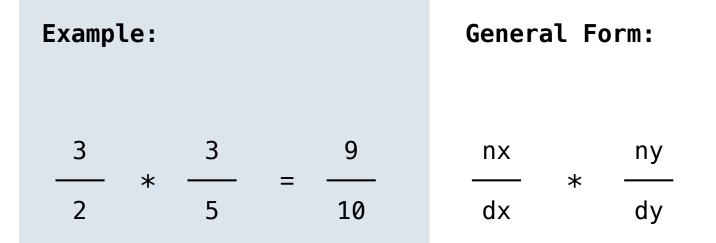
General Form:

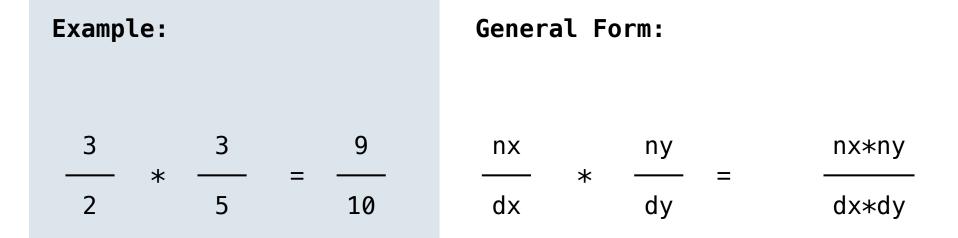
Example:

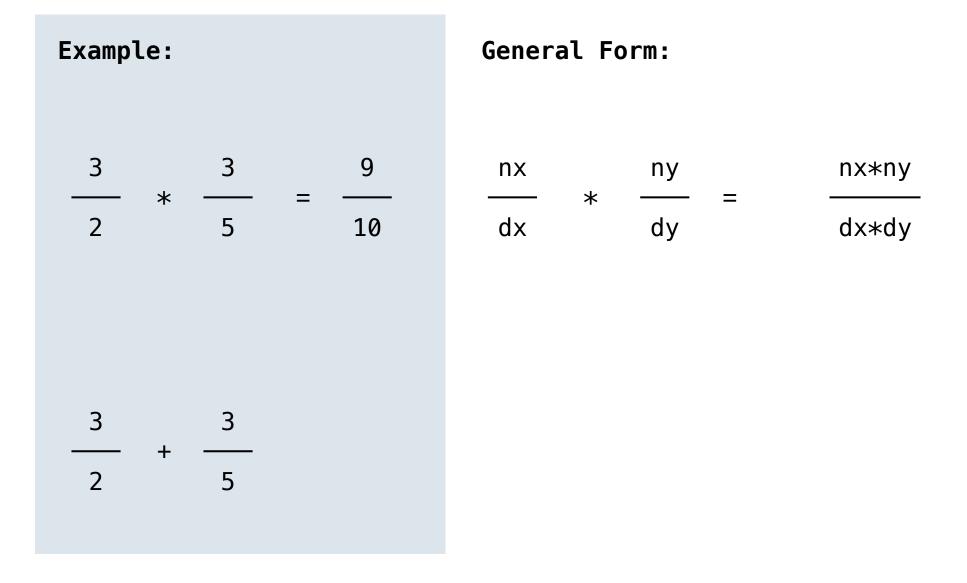
$$\frac{3}{2} * \frac{3}{5}$$

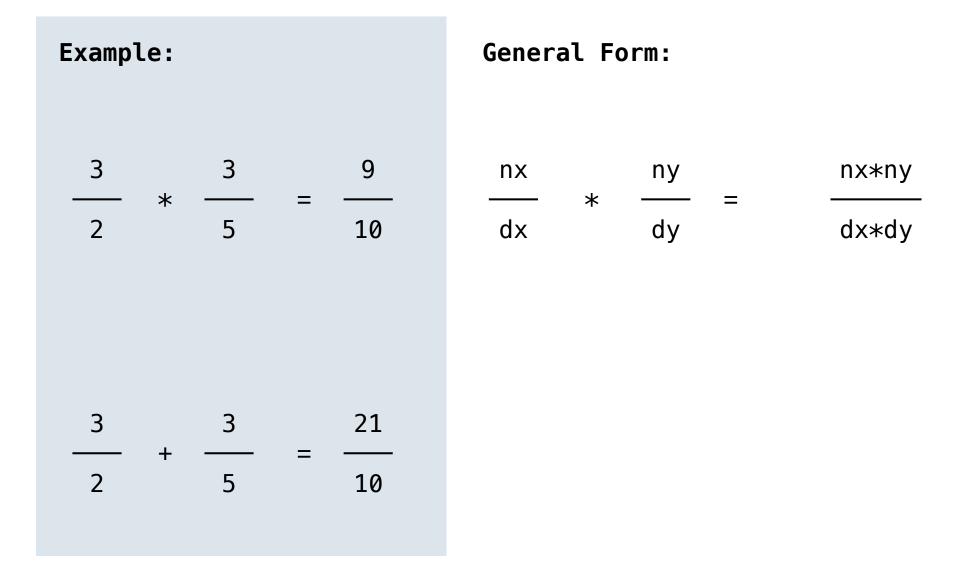
General Form:

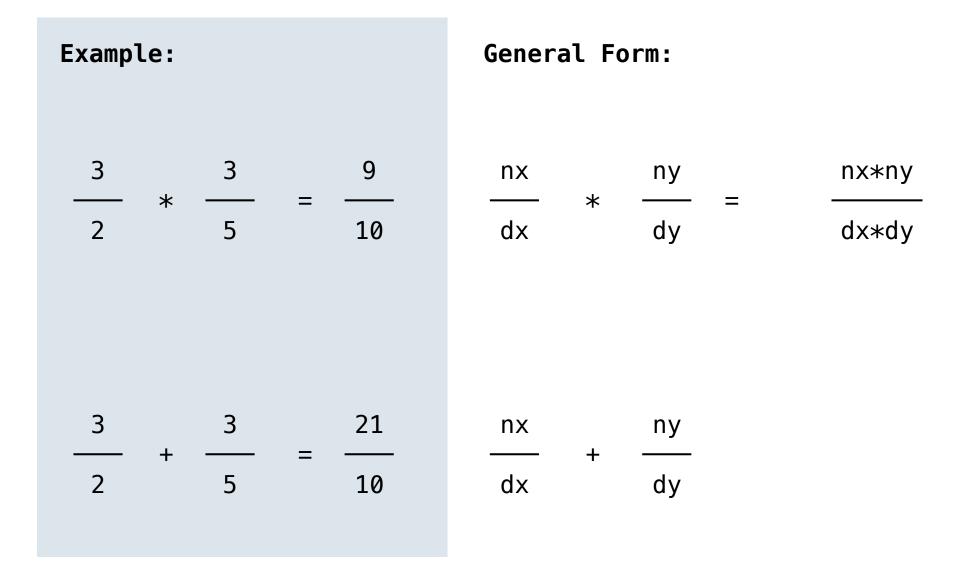
Example: $\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$ General Form:

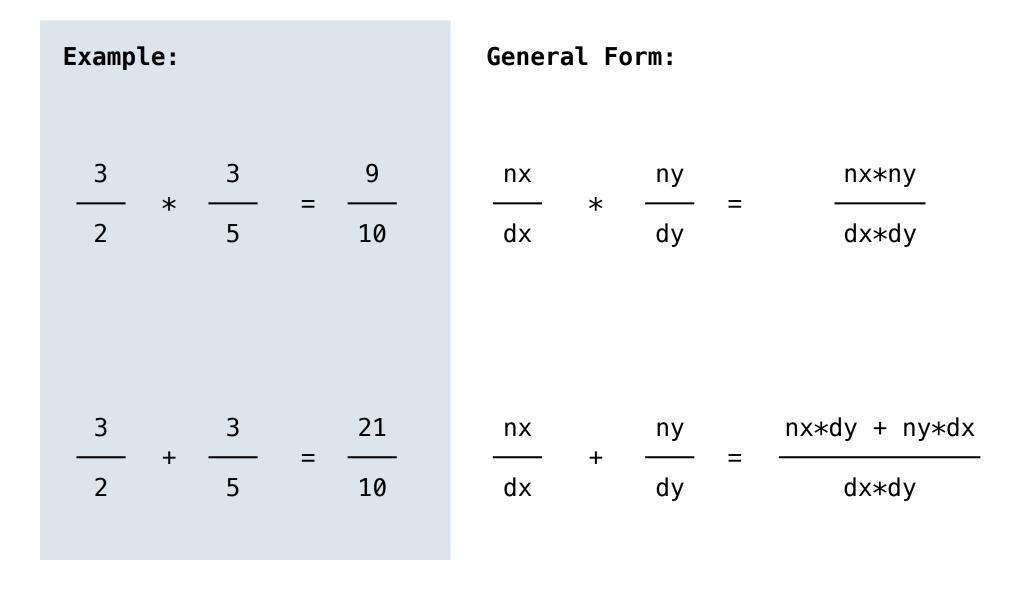




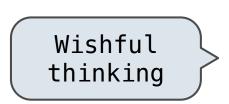






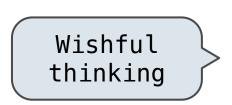


- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x



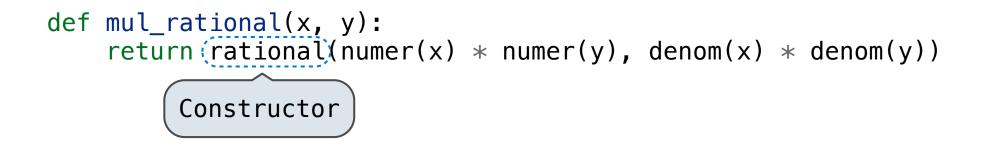
- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

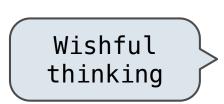
```
def mul_rational(x, y):
    return rational(numer(x) * numer(y), denom(x) * denom(y))
```



- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

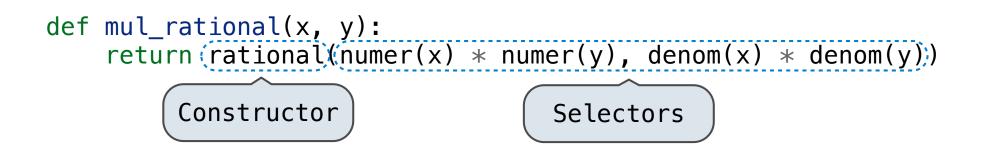
Rational Number Arithmetic Implementation

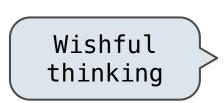




- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

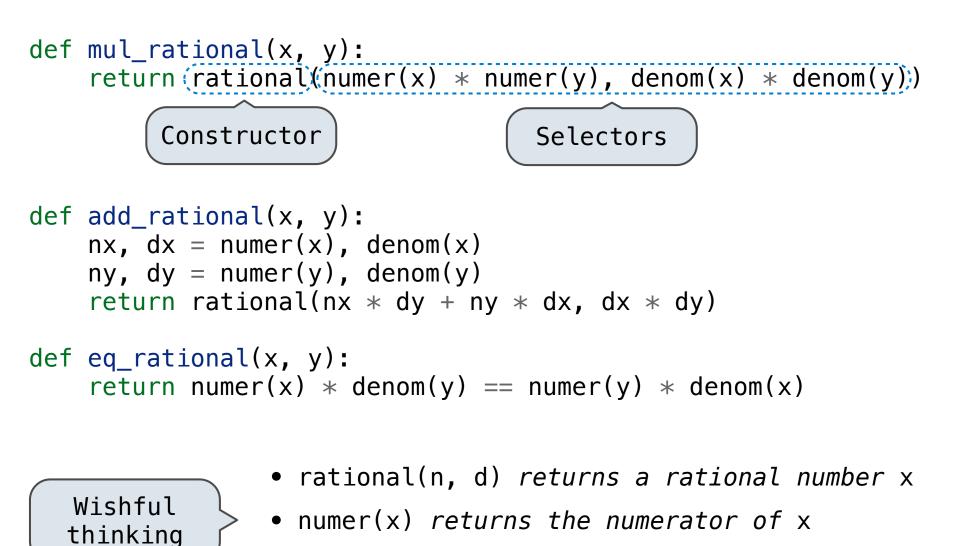
Rational Number Arithmetic Implementation





- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

Rational Number Arithmetic Implementation



• denom(x) returns the denominator of x

>>> pair = (1, 2)

>>> pair = (1, 2) >>> pair

```
>>> pair = (1, 2)
>>> pair
(1, 2)
```

```
>>> pair = (1, 2)
>>> pair
(1, 2)
```

A tuple literal: Comma-separated expression

>>> pair = (1, 2) >>> pair (1, 2)

A tuple literal: Comma-separated expression

>>> x, y = pair

>>> pair = (1, 2) >>> pair (1, 2)

A tuple literal: Comma-separated expression

>>> x, y = pair >>> x

```
>>> pair = (1, 2)
>>> pair
(1, 2)
```

>>> x, y = pair >>> x 1 A tuple literal: Comma-separated expression

>>> pair = (1, 2) >>> pair (1, 2)

A tuple literal: Comma-separated expression

```
>>> x, y = pair
>>> x
1
>>> y
```

>>> y 2

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> x
1
```

A tuple literal: Comma-separated expression

6

>>> pair = (1, 2) >>> pair (1, 2)

```
>>> x, y = pair
>>> x
1
>>> y
2
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
```

```
>>> x, y = pair
>>> x
1
>>> y
2
```

A tuple literal: Comma-separated expression

"Unpacking" a tuple

>>> pair[0]

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> x
1
>>> y
2
```

A tuple literal: Comma-separated expression

```
>>> pair[0]
1
```

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> x
1
>>> y
2
>>> pair[0]
```

A tuple literal: Comma-separated expression

```
>>> pair[0]
1
>>> pair[1]
```

2

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> x
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A tuple literal: Comma-separated expression

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A tuple literal: Comma-separated expression

"Unpacking" a tuple

Element selection

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> X
1
>>> y
2
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A tuple literal: Comma-separated expression

"Unpacking" a tuple

Element selection

```
More tuples next lecture
```

```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)
```

```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return((n, d))
    Construct a tuple
```

```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return((n, d))
    Construct a tuple
```

from operator import getitem

```
def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)
```

```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return((n, d))
    Construct a tuple
```

from operator import getitem

```
def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)
```

def denom(x):
 """Return the denominator of rational number x."""
 return getitem(x, 1)

```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return((n, d))
    Construct a tuple
```

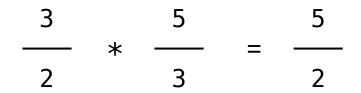
from operator import getitem

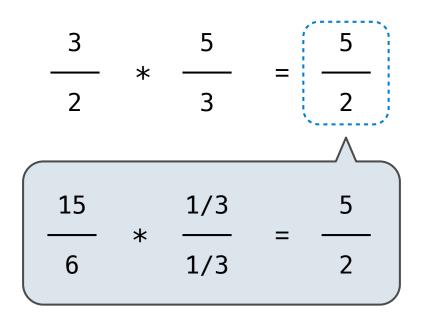
```
def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)
```

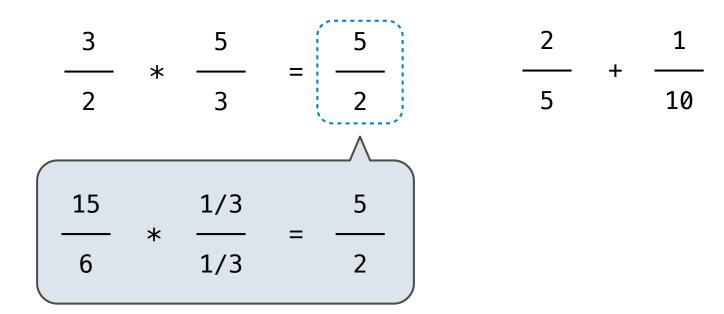
def denom(x):
 """Return the denominator of rational number x."""
 return(getitem(x, 1))

Select from a tuple

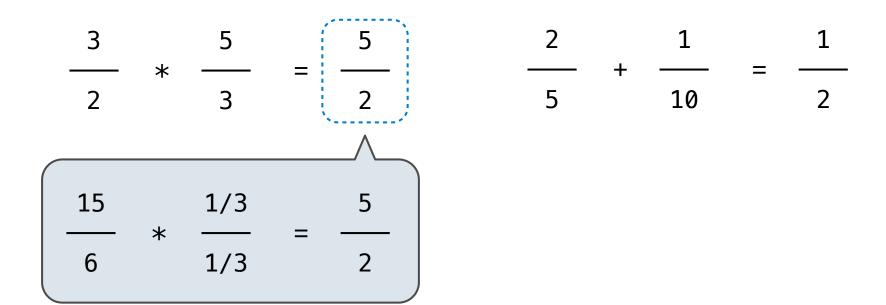




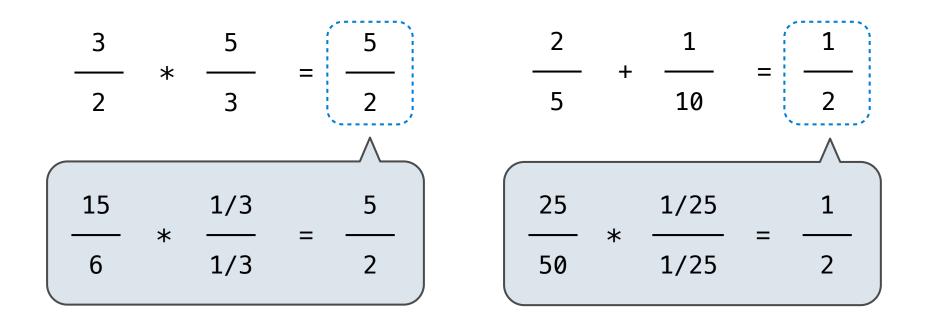




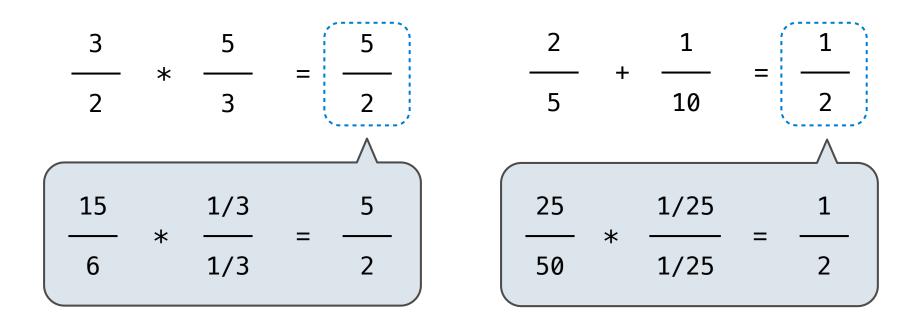
Example:



Example:

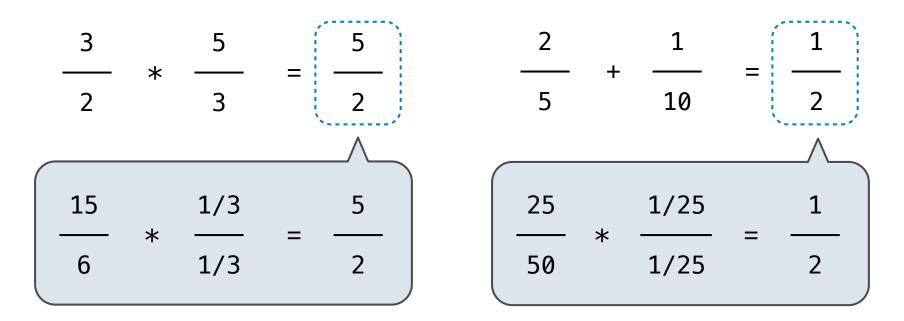


Example:



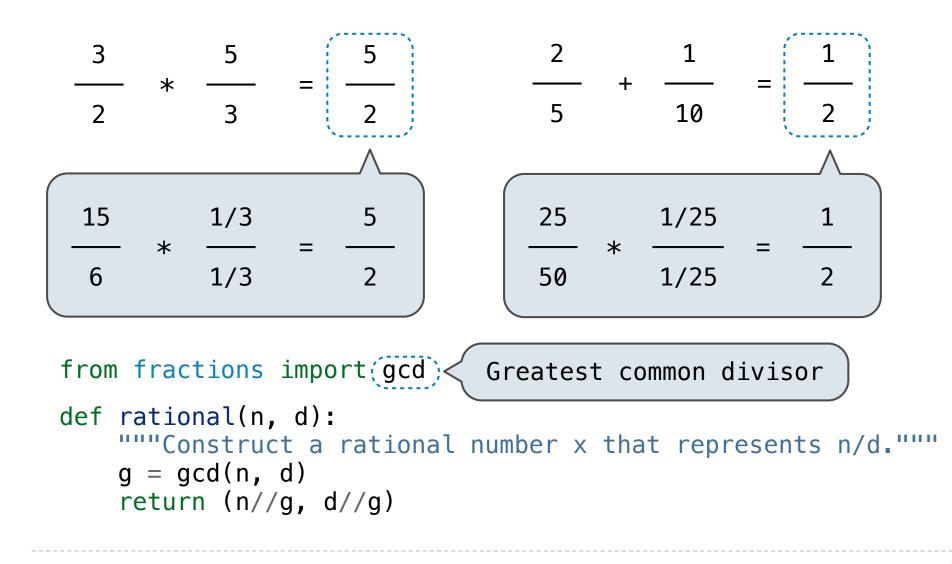
from fractions import gcd

Example:



from fractions import(gcd) < Greatest common divisor</pre>

Example:



Rational numbers as whole data values

add_rationals mul_rationals eq_rationals

Rational numbers as numerators & denominators

rational numer denom

Rational numbers as tuples

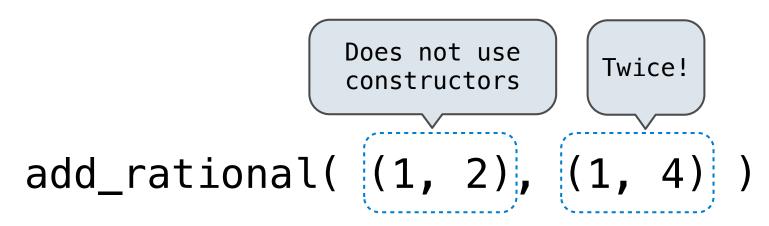
tuple getitem

However tuples are implemented in Python

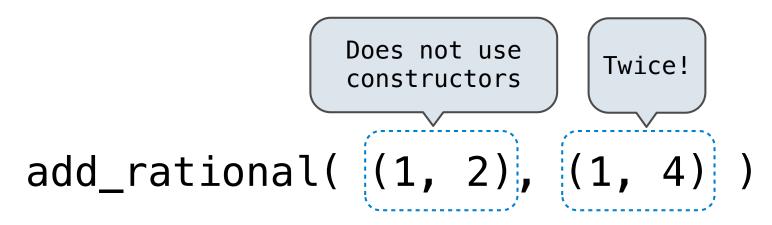
add_rational((1, 2), (1, 4))

def divide_rational(x, y): return (x[0] * y[1], x[1] * y[0])

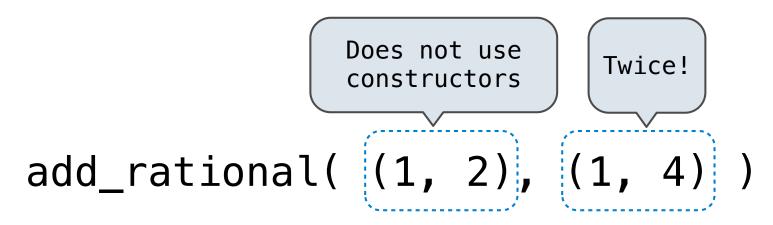
def divide_rational(x, y): return (x[0] * y[1], x[1] * y[0])

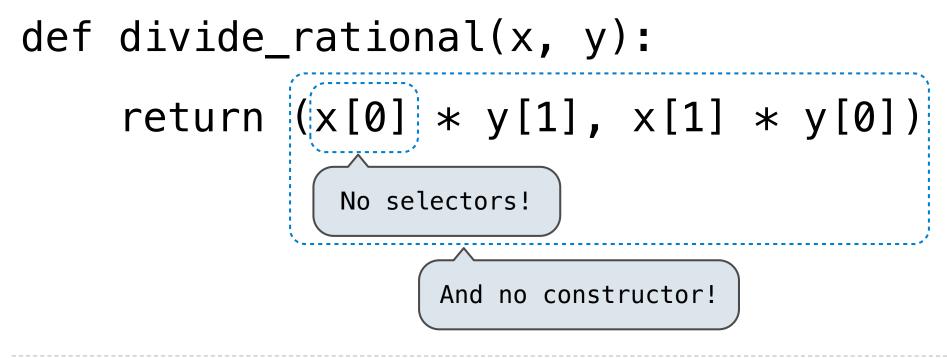


def divide_rational(x, y): return (x[0] * y[1], x[1] * y[0])



def divide_rational(x, y): return (x[0] * y[1], x[1] * y[0]) No selectors!







• We need to guarantee that constructor and selector functions together specify the right behavior.

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Behavior condition: If we construct rational number x from numerator n and denominator d, then numer(x)/denom(x) must equal n/d.

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Behavior condition: If we construct rational number x from numerator n and denominator d, then numer(x)/denom(x) must equal n/d.
- An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Behavior condition: If we construct rational number x from numerator n and denominator d, then numer(x)/denom(x) must equal n/d.
- An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).
- If behavior conditions are met, the representation is valid.

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Behavior condition: If we construct rational number x from numerator n and denominator d, then numer(x)/denom(x) must equal n/d.
- An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).
- If behavior conditions are met, the representation is valid.

You can recognize data types by behavior, not by bits

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

Constructors, selectors, and behavior conditions:

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

- getitem_pair(p, 0) returns x, and
- getitem_pair(p, 1) returns y.

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

- getitem_pair(p, 0) returns x, and
- getitem_pair(p, 1) returns y.

Together, selectors are the inverse of the constructor

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

- getitem_pair(p, 0) returns x, and
- getitem_pair(p, 1) returns y.

Together, selectors are the inverse of the constructor Generally true of *container types*.

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

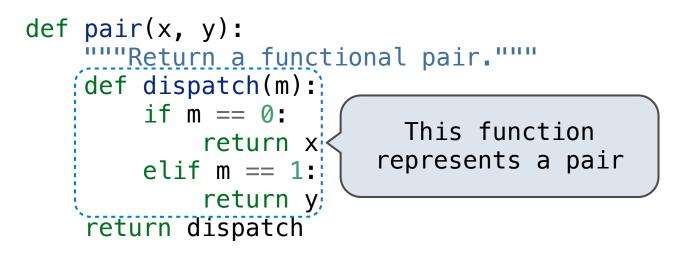
- getitem_pair(p, 0) returns x, and
- getitem_pair(p, 1) returns y.

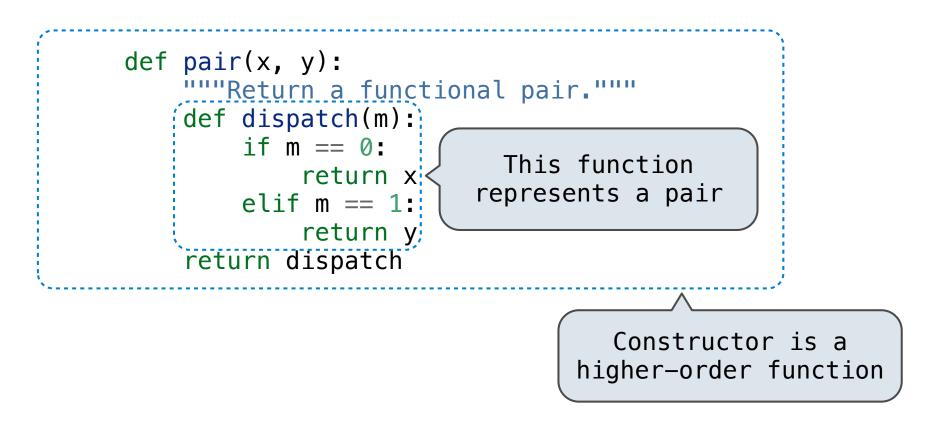
Together, selectors are the inverse of the constructor

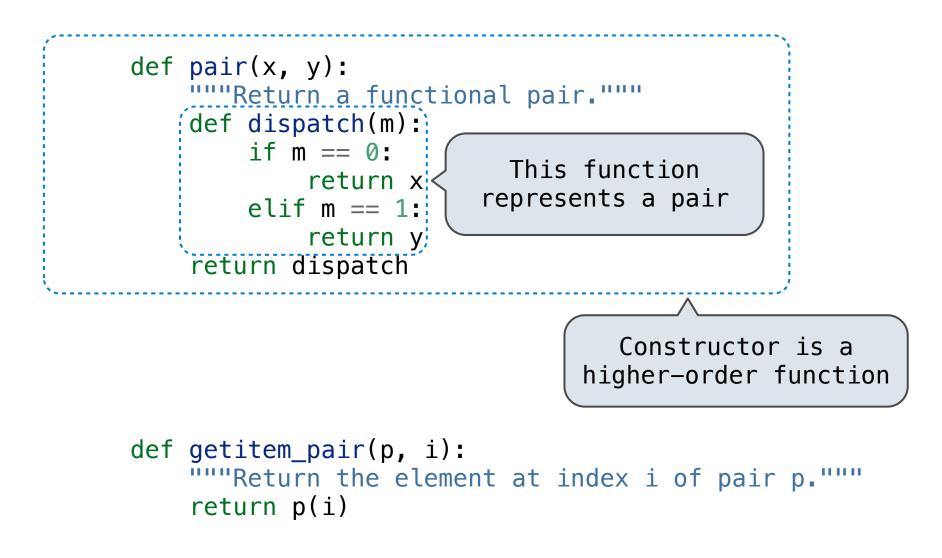
Generally true of container types.

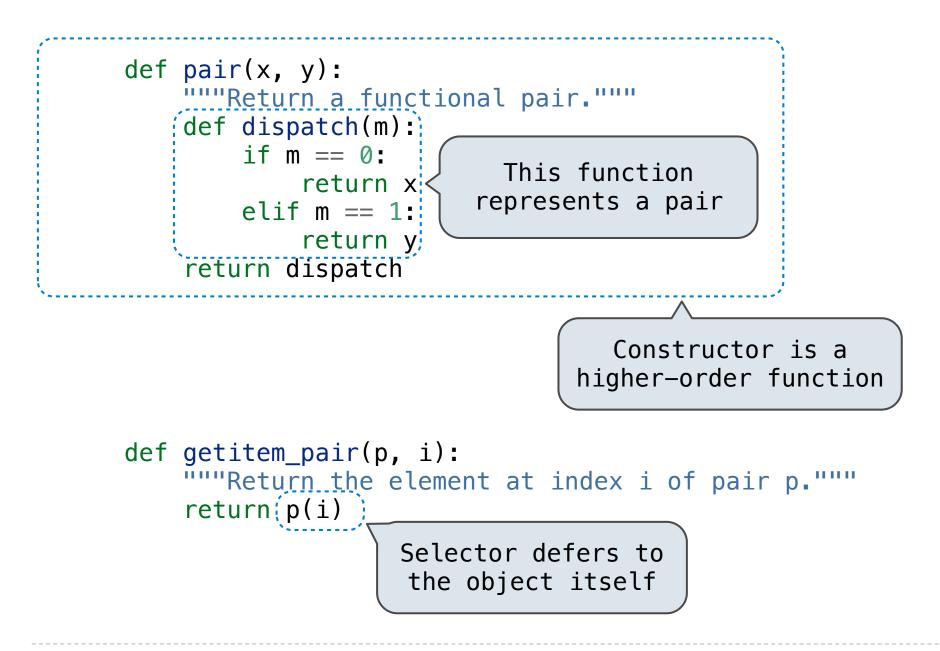
Not true for rational numbers because of GCD

```
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```









```
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
```

```
>>> p = pair(1, 2)
```

```
>>> getitem_pair(p, 0)
1
```

```
>>> getitem_pair(p, 1)
2
```

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions

```
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
```

```
>>> getitem_pair(p, 1)
2
```

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions

If a pair p was constructed from elements x and y, then

- getitem_pair(p, 0) returns x, and
- getitem_pair(p, 1) returns y.

```
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
```

```
>>> getitem_pair(p, 1)
2
```

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions

If a pair p was constructed from elements x and y, then

- getitem_pair(p, 0) returns x, and
- getitem_pair(p, 1) returns y.

This pair representation is valid!