

61A Lecture 8

Wednesday, September 14

Data Abstraction

Data Abstraction

- Compound objects combine primitive objects together

Data Abstraction

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

Data Abstraction

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

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

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:

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)

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

- 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***

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

Rational Numbers

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

A pair of integers

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

A pair of integers

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

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

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 Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

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:

- `make_rat(n, d)` *returns a rational number x*

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

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 `make_rat(n, d)` returns a rational number x

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

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

`make_rat(n, d)` returns a rational number x

- `numer(x)` returns the numerator of x

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

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

`make_rat(n, d)` returns a rational number x

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

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

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

`make_rat(n, d)` returns a rational number x

Selectors

• `numer(x)` returns the numerator of x

• `denom(x)` returns the denominator of x

Rational Number Arithmetic

Example:

General Form:

Rational Number Arithmetic

Example:

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

General Form:

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

General Form:

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy}$$

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

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

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy}$$

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

Rational Number Arithmetic Implementation

Rational Number Arithmetic Implementation

- `make_rat(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

Wishful
thinking

- `make_rat(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

```
def mul_rat(x, y):  
    """Multiply rational numbers x and y."""  
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

Wishful
thinking

- `make_rat(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

```
def mul_rat(x, y):  
    """Multiply rational numbers x and y."""  
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

Constructor

Wishful
thinking

- `make_rat(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

```
def mul_rat(x, y):  
    """Multiply rational numbers x and y."""  
    return make_rat( numer(x) * numer(y), denom(x) * denom(y) )
```

Constructor

Selectors

Wishful
thinking

- `make_rat(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

```
def mul_rat(x, y):  
    """Multiply rational numbers x and y."""  
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

Constructor

Selectors

```
def add_rat(x, y):  
    """Add rational numbers x and y."""  
    nx, dx = numer(x), denom(x)  
    ny, dy = numer(y), denom(y)  
    return make_rat(nx * dy + ny * dx, dx * dy)
```

Wishful
thinking

- `make_rat(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

```
def mul_rat(x, y):  
    """Multiply rational numbers x and y."""  
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

Constructor

Selectors

```
def add_rat(x, y):  
    """Add rational numbers x and y."""  
    nx, dx = numer(x), denom(x)  
    ny, dy = numer(y), denom(y)  
    return make_rat(nx * dy + ny * dx, dx * dy)
```

```
def eq_rat(x, y):  
    """Return whether rational numbers x and y are equal."""  
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Wishful
thinking

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

Tuples

Tuples

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

Tuples

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

Tuples

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

Tuples

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

A tuple literal:
Comma-separated expressions

Tuples

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

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

A tuple literal:
Comma-separated expressions

Tuples

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

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

A tuple literal:
Comma-separated expressions

Tuples

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

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

A tuple literal:
Comma-separated expressions

Tuples

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

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

A tuple literal:
Comma-separated expressions

Tuples

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

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

A tuple literal:
Comma-separated expressions

Tuples

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

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

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Tuples

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

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

```
>>> from operator import getitem
```

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Tuples

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

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

```
>>> from operator import getitem
>>> getitem(pair, 0)
```

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Tuples

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

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

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
```

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Tuples

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

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

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
```

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Tuples

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

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

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Tuples

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

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

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Element selection

Tuples

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

A tuple literal:
Comma-separated expressions

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

"Unpacking" a tuple

```
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

Element selection

More tuples next lecture

Representing Rational Numbers

Representing Rational Numbers

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

Representing Rational Numbers

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

Construct a tuple

Representing Rational Numbers

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

Construct a tuple

```
from operator import getitem
```

Representing Rational Numbers

```
def make_rat(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)
```

Representing Rational Numbers

```
def make_rat(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

Reducing to Lowest Terms

Example:

Reducing to Lowest Terms

Example:

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

Reducing to Lowest Terms

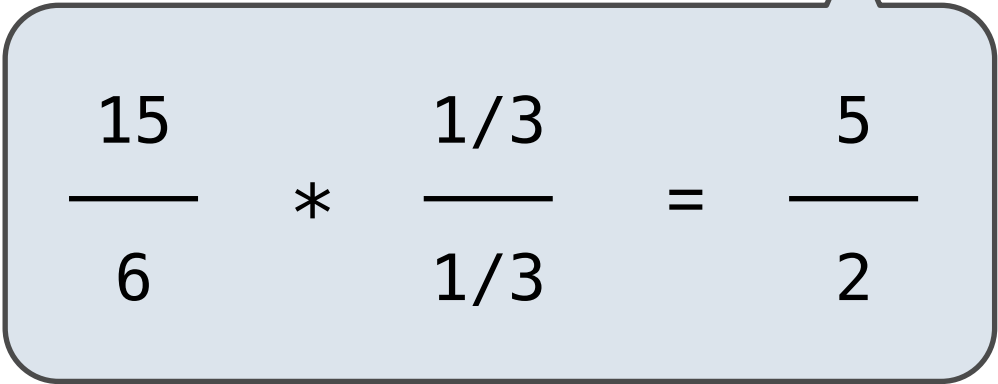
Example:

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

Reducing to Lowest Terms

Example:

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


$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

Reducing to Lowest Terms

Example:

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

$$\frac{2}{5} + \frac{1}{10}$$

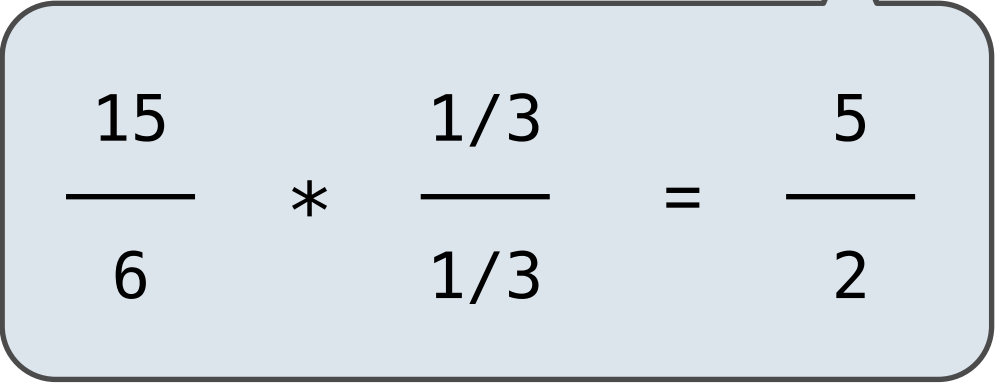
$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

Reducing to Lowest Terms

Example:

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

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$


$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

Reducing to Lowest Terms

Example:

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

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

Reducing to Lowest Terms

Example:

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

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd
def make_rat(n, d):
    """Construct a rational number x that represents n/d in lowest terms."""
    g = gcd(n, d)
    return (n//g, d//g)
```


Reducing to Lowest Terms

Example:

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

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

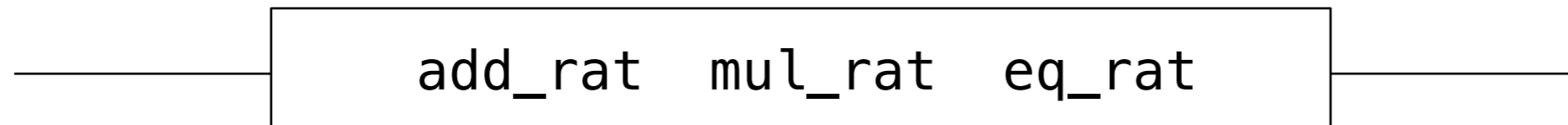
```
from fractions import gcd
def make_rat(n, d):
    """Construct a rational number x that represents n/d in lowest terms."""
    g = gcd(n, d)
    return (n//g, d//g)
```

Greatest common divisor

Abstraction Barriers

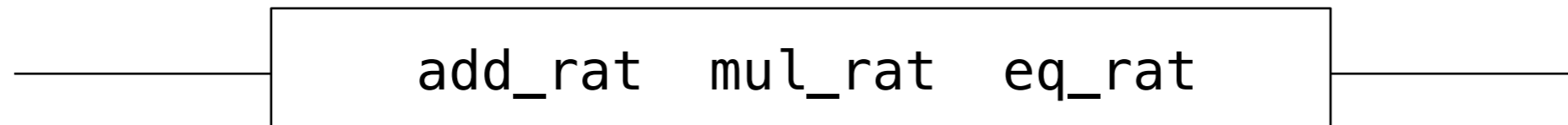
Abstraction Barriers

Rational numbers in the problem domain

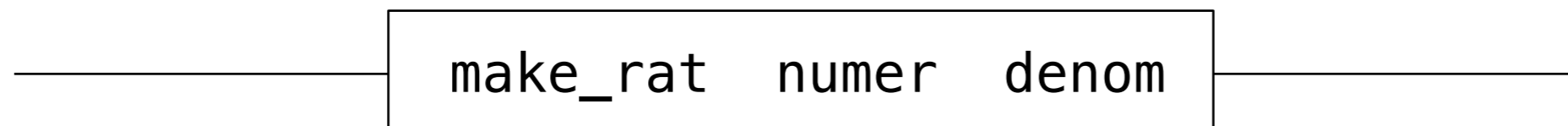


Abstraction Barriers

Rational numbers in the problem domain



Rational numbers as numerators & denominators



Abstraction Barriers

Rational numbers in the problem domain

add_rat mul_rat eq_rat

Rational numbers as numerators & denominators

make_rat numer denom

Rational numbers as tuples

tuple getitem

Abstraction Barriers

Rational numbers in the problem domain

add_rat mul_rat eq_rat

Rational numbers as numerators & denominators

make_rat numer denom

Rational numbers as tuples

tuple getitem

However tuples are implemented in Python

Violating Abstraction Barriers

```
add_rat( (1, 2), (1, 4) )
```

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

Violating Abstraction Barriers

Does not use
constructors

```
add_rat( (1, 2), (1, 4) )
```

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


Violating Abstraction Barriers

Does not use
constructors

Twice!

```
add_rat( (1, 2), (1, 4) )
```

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

Violating Abstraction Barriers

Does not use
constructors

Twice!

```
add_rat( (1, 2), (1, 4) )
```

```
def divide_rat(x, y):
```

```
    return (x[0] * y[1], x[1] * y[0])
```

No selectors!

Violating Abstraction Barriers

Does not use
constructors

Twice!

```
add_rat( (1, 2), (1, 4) )
```

```
def divide_rat(x, y):
```

```
    return (x[0] * y[1], x[1] * y[0])
```

No selectors!

And no constructor!

Violating Abstraction Barriers

What is Data?

What is Data?

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

What is Data?

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Rational numbers: If we construct x from n and d , then $\text{numer}(x)/\text{denom}(x)$ must equal n/d .

What is Data?

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Rational numbers: If we construct x from n and d , then $\text{numer}(x)/\text{denom}(x)$ must equal n/d .
- An abstract data type is some collection of selectors and constructors, together with some behavior conditions.

What is Data?

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Rational numbers: If we construct x from n and d , then $\text{numer}(x)/\text{denom}(x)$ must equal n/d .
- An abstract data type is some collection of selectors and constructors, together with some behavior conditions.
- If behavior conditions are met, the representation is valid.

What is Data?

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Rational numbers: If we construct x from n and d , then $\text{numer}(x)/\text{denom}(x)$ must equal n/d .
- An abstract data type is some collection of selectors and constructors, together with some behavior conditions.
- If behavior conditions are met, the representation is valid.

You can recognize data types by behavior, not by bits

Behavior Conditions of a Pair

Behavior Conditions of a Pair

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

Behavior Conditions of 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?

Behavior Conditions of 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:

Behavior Conditions of 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:

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

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

Behavior Conditions of 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:

If a pair `p` was constructed from values `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

Behavior Conditions of 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:

If a pair `p` was constructed from values `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*.

Behavior Conditions of 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:

If a pair `p` was constructed from values `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

Functional Pair Implementation

Functional Pair Implementation

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

Functional Pair Implementation

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

This function represents a pair

Functional Pair Implementation

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

This function represents a pair

Constructor is a higher-order function

Functional Pair Implementation

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

This function represents a pair

Constructor is a higher-order function

```
def getitem_pair(p, i):  
    """Return the element at index i of pair p."""  
    return p(i)
```

Functional Pair Implementation

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

This function represents a pair

Constructor is a higher-order function

```
def getitem_pair(p, i):  
    """Return the element at index i of pair p."""  
    return p(i)
```

Selector defers to the object itself

Using a Functionally Implemented Pair

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

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

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

Using a Functionally Implemented Pair

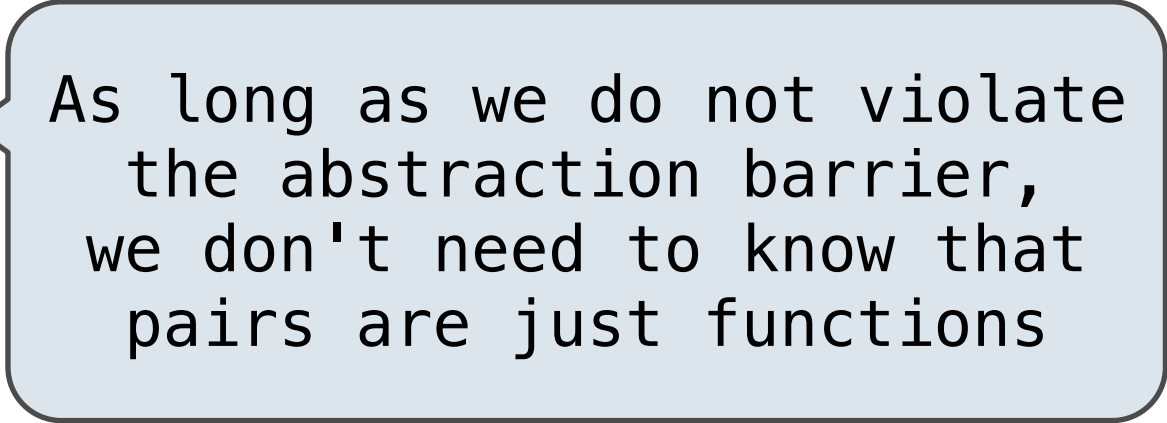
```
>>> p = make_pair(1, 2)
```

```
>>> 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

Using a Functionally Implemented Pair

```
>>> p = make_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 values `x` and `y`, then

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

Using a Functionally Implemented Pair

```
>>> p = make_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 values `x` and `y`, then

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

This pair representation is valid!