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
Rational Numbers
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]

Exact representation of fractions
Rational Numbers

\[
\begin{array}{c}
\text{numerator} \\
\hline
\text{denominator}
\end{array}
\]

Exact representation of fractions

A pair of integers
Rational Numbers

numerator

-----------------
denominator

Exact representation of fractions

A pair of integers

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

\[
\begin{array}{c}
\text{numerator} \\
\hline \\
\text{denominator}
\end{array}
\]

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
Rational Numbers

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\[\text{make\_rat(n, d)}\] \text{returns a rational number}
Rational Numbers

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:

- \texttt{make\_rat(n, d)} \textit{returns a rational number} \( x \)
- \texttt{numer(x)} \textit{returns the numerator of} \( x \)
Rational Numbers

\[
\begin{array}{c}
\text{numerator} \\
\hline
\text{denominator}
\end{array}
\]

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:

- \textbf{Constructor}: \texttt{make_rat(n, d)} \textit{returns a rational number} \( x \)
- \texttt{numer(x)} \textit{returns the numerator of} \( x \)
- \texttt{denom(x)} \textit{returns the denominator of} \( x \)
Rational Numbers

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:

\[
\frac{3}{2} \times \frac{3}{5}
\]

General Form:
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

General Form:
Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy}
\]
Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5} = \frac{21}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx*ny}{dx*dy}
\]
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5} = \frac{21}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]

\[
\frac{nx}{dx} + \frac{ny}{dy}
\]
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5} = \frac{21}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]

\[
\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx \times dy + ny \times dx}{dx \times dy}
\]
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

- \textbf{make\_rat}(n, d) \textit{returns a rational number }x
- \textbf{numer}(x) \textit{returns the numerator of }x
- \textbf{denom}(x) \textit{returns the denominator of }x
def mul_rat(x, y):
    """Multiply rational numbers x and y.""
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))

- 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

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

- `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

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

- `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

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

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

Wishful thinking
Rational Number Arithmetic Implementation

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

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

- `make_rat(n, d)` returns a rational number \( \frac{n}{d} \)
- `numer(x)` returns the numerator of \( x \)
- `denom(x)` returns the denominator of \( x \)
Tuples
Tuples

```python
>>> 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

Sunday, September 11, 2011
Tuples

A tuple literal:
Comma-separated expressions

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

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

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

>>> x, y = pair
>>> x
1
>>> y
Tuples

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

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

A tuple literal:
Comma-separated expressions
Tuples

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

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

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

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple
Tuples

```python
>>> 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

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

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

>>> x, y = pair

>>> x
1

>>> y
2

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

```python
>>> 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

A tuple literal:
Comma-separated expressions

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

"Unpacking" a tuple

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

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

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

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

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

A tuple literal:
Comma-separated expressions

"Unpacking" a tuple

Element selection
Tuples

```python
>>> 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

More tuples next lecture
Representing Rational Numbers
Representing Rational Numbers

def make_rat(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

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

```python
from operator import getitem
```
Representing Rational Numbers

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

from operator importgetitem

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):
    
    return (n, d)

Construct a tuple

from operator importgetitem

def numer(x):
    
    returngetitem(x, 0)

def denom(x):
    
    returngetitem(x, 1)

Select from a tuple
Reducing to Lowest Terms

Example:
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]

\[
\frac{2}{5} + \frac{1}{10}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]

\[
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]

\[
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]

\[
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]

\[
\frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

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:

\[
\begin{align*}
\frac{3}{2} \times \frac{5}{3} &= \frac{5}{2} \\
\frac{15}{6} \times \frac{1/3}{1/3} &= \frac{5}{2} \\
\frac{2}{5} + \frac{1}{10} &= \frac{1}{2} \\
\frac{25}{50} \times \frac{1/25}{1/25} &= \frac{1}{2}
\end{align*}
\]

```python
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
Rational numbers in the problem domain

add_rat  mul_rat  eq_rat
Abstraction Barriers

Rational numbers in the problem domain

- add_rat
- mul_rat
- eq_rat

Rational numbers as numerators & denominators

- make_rat
- numer
- denom

Sunday, September 11, 2011
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])
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])
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

### add_rat

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

### divide_rat

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

- Does not use constructors
- Twice!
- 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 numer(x)/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 $\text{numerator}(x)/\text{denominator}(x)$ 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:
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

- \( \text{getitem}_\text{pair}(p, 0) \) returns \( x \), and
- \( \text{getitem}_\text{pair}(p, 1) \) 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

- \( \text{getitem\_pair}(p, 0) \) returns \( x \), and
- \( \text{getitem\_pair}(p, 1) \) 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

- \( \text{getitem}_\text{pair}(p, 0) \) returns \( x \), and
- \( \text{getitem}_\text{pair}(p, 1) \) returns \( y \).

Together, selectors are the inverse of the constructor.

Generally true of container types. Not true for rational numbers.
Functional Pair Implementation
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

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

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

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

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

This function represents a pair

Constructor is a higher-order function

Selector defers to the object itself
Using a Functionally Implemented Pair

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

>>> getitem_pair(p, 0)
1

>>> getitem_pair(p, 1)
2
```
Using a Functionally Implemented Pair

```python
>>> 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

```python
>>> 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

```python
d>>> p = make_pair(1, 2)
d>>> getitem_pair(p, 0)
1
d>>> 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!