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

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 \( x \)

- **Selectors**
  - `numer(x)` returns the numerator of \( x \)
  - `denom(x)` returns the denominator of \( x \)
### Rational Number Arithmetic

**Example:**

\[
\begin{array}{ccc}
\frac{3}{2} & \times & \frac{3}{5} \\
\hline \\
& & = \\
\frac{9}{10}
\end{array}
\]

\[
\begin{array}{ccc}
\frac{3}{2} & + & \frac{3}{5} \\
\hline \\
& & = \\
\frac{21}{10}
\end{array}
\]

**General Form:**

\[
\begin{array}{ccc}
\frac{nx}{dx} & \times & \frac{ny}{dy} \\
\hline \\
& & = \\
\frac{nx \times ny}{dx \times dy}
\end{array}
\]

\[
\begin{array}{ccc}
\frac{nx}{dx} & + & \frac{ny}{dy} \\
\hline \\
& & = \\
\frac{nx \times dy + ny \times dx}{dx \times dy}
\end{array}
\]
Rational Number Arithmetic Implementation

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

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)
```

```
def eq_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

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

```python
>>> 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)

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)
Reducing to Lowest Terms

Example:

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

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

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

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

```python
from fractions import gcd

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

Greatest common divisor
Abstraction Barriers

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
Violating Abstraction Barriers

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

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

No selectors!
And no constructor!
What is Data?

• 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*
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 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

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

```python
>>> 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!