Data Abstraction

- Compound values combine other values together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- Data abstraction lets us manipulate compound values 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

- **numerator**
- **denominator**

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation may be lost! (Demo)

Assume we can compose and decompose rational numbers:

- **Constructor**: `rational(a, d)` returns a rational number `x`
- **Selectors**:
  - `numer(x)` returns the numerator of `x`
  - `denom(x)` returns the denominator of `x`

Rational Number Arithmetic

<table>
<thead>
<tr>
<th></th>
<th>numerator</th>
<th>denominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/2</td>
<td>3/5</td>
<td>9/10</td>
</tr>
<tr>
<td>3/2</td>
<td>3/5</td>
<td>21/10</td>
</tr>
</tbody>
</table>

General Form

Example

```
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 print_rational(x):
    print(numer(x), '/', denom(x))

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

Pairs
Representing Pairs Using Lists

```python
>>> pair = [1, 2]  # A list literal: Comma-separated expressions in brackets
>>> x, y = pair     # "Unpacking" a list
>>> x
1
>>> y
2
>>> pair[0]        # Element selection using the selection operator
1
>>> pair[1]
2
>>> from operator import getitem  # Element selection function
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

Representing Rational Numbers

```python
def rational(n, d):
    return [n, d]  # Construct a rational number that represents N/D.

def numerator():
    return x[0]  # Return the numerator of rational number X.

def denominator():
    return x[1]  # Return the denominator of rational number X.
```

Reducing to Lowest Terms

```python
from fractions import gcd

def rational(n, d):
    g = gcd(n, d)  # Greatest common divisor
    return [n // g, d // g]  # Construct a rational that represents n/d in lowest terms.
```

Abstraction Barriers

<table>
<thead>
<tr>
<th>Parts of the program that...</th>
<th>Treat rationals as...</th>
<th>Use...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use rational numbers to perform computation</td>
<td>whole data values</td>
<td>add_rational, mul_rational, rational_are_equal, print_rational</td>
</tr>
<tr>
<td>Create rational or implement rational operations</td>
<td>numerators and denominators</td>
<td>rational, numerator, denominator</td>
</tr>
<tr>
<td>Implement selectors and constructor for rationals</td>
<td>two-element lists</td>
<td>list literals and element selection</td>
</tr>
</tbody>
</table>

Violating Abstraction Barriers

```python
add_rational([1, 2], [1, 4])  # Does not use constructors twice!

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

What is Data?

- We need to guarantee that constructor and selector functions work together to specify the right behavior.
- Behavior condition: If we construct rational number x from numerator n and denominator d, then `numerator(x)/denominator(x)` must equal n/d.
- Data abstraction uses selectors and constructors to define behavior.
- If behavior conditions are met, then the representation is valid.

You can recognize an abstract data representation by its behavior.
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
        return select
    return select

def numer(x):
    return x('n')

def denom(x):
    return x('d')

This function represents a rational number.

Rationals Implemented as Functions

Interactive Diagram

x = rational(3, 8)
numer(x)