Data Abstraction
Announcements
Data Abstraction
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

A pair of integers

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

Assume we can compose and decompose rational numbers:

- `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

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

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

Example

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

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

General Form
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 print_rational(x):
    print(numer(x), '/', denom(x))

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

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

These functions implement an abstract representation for rational numbers.
Representing Rational Numbers
Representing Pairs Using Lists

>>> pair = [1, 2]  # A list literal: Comma-separated expressions in brackets
>>> pair
[1, 2]

>>> x, y = pair  # "Unpacking" a list
>>> x
1
>>> y
2

>>> pair[0]  # Element selection using the selection operator
1
>>> pair[1]
2
Representing Rational Numbers

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

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

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

(Demo)
Reducing to Lowest Terms

Example:

\[
\begin{align*}
\frac{3}{2} \times \frac{5}{3} &= \frac{5}{2} \\
\frac{2}{5} + \frac{1}{10} &= \frac{1}{2}
\end{align*}
\]

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

```python
from math import gcd

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

(Demo)
Abstraction Barriers
### Abstraction Barriers

<table>
<thead>
<tr>
<th>Parts of the program that...</th>
<th>Treat rationals as...</th>
<th>Using...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use rational numbers to perform computation</td>
<td>whole data values</td>
<td>add_rational, mul_rational, rationals_are_equal, print_rational</td>
</tr>
<tr>
<td>Create rationals or implement rational operations</td>
<td>numerators and denominators</td>
<td>rational, numer, denom</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>

**Implementation of lists**
Violating Abstraction Barriers

add_rational( [1, 2], [1, 4] )

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

- Does not use constructors
- Twice!
- No selectors!
- And no constructor!
Data Representations
What are 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 numer(x)/denom(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

(Demo)
Rationals Implemented as Functions

```python
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

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

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

def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

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

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

This function represents a rational number

Constructor is a higher-order function

Selector calls x

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

{"Dem": 0}
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)

- Two keys **cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value
Dictionary Comprehensions

\{<\text{key exp}>: <\text{value exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>\}\n
Short version: \{<\text{key exp}>: <\text{value exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}>\}\n
An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty \textit{result dictionary} that is the value of the expression
3. For each element in the iterable value of <\text{iter exp}>:
   A. Bind <\text{name}> to that element in the new frame from step 1
   B. If <\text{filter exp}> evaluates to a true value, then add to the result dictionary an entry that pairs the value of <\text{key exp}> to the value of <\text{value exp}>

\{x \ast x: x \text{ for } x \text{ in } [1, 2, 3, 4, 5] \text{ if } x > 2\} \text{ evaluates to } \{9: 3, 16: 4, 25: 5\}
Example: Indexing

Implement `index`, which takes a sequence of `keys`, a sequence of `values`, and a two-argument `match` function. It returns a dictionary from `keys` to lists in which the list for a key `k` contains all `values` `v` for which `match(k, v)` is a true value.

```python
def index(keys, values, match):
    """Return a dictionary from keys `k` to a list of values `v` for which `match(k, v)` is a true value.
    """

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return {k: [v for v in values if match(k, v)] for k in keys}
```