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***

All Programmers

Great Programmers

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(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

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

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

Example

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

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

General Form

Rational Number Arithmetic Implementation

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

Constructor

Selectors

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

```
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)
```

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

- rational(n, d) returns a rational number x
- 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]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

```
>>> pair[0]
1
>>> pair[1]
2
```

Element selection using the selection operator

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

Element selection function

Representing Rational Numbers

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

Construct a list

```
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]
```

Select item from a list

(Demo)

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from math import gcd
```

Greatest common divisor

```
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

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Implementation of lists

Violating Abstraction Barriers

Does not use constructors

Twice!

```
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!

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
```

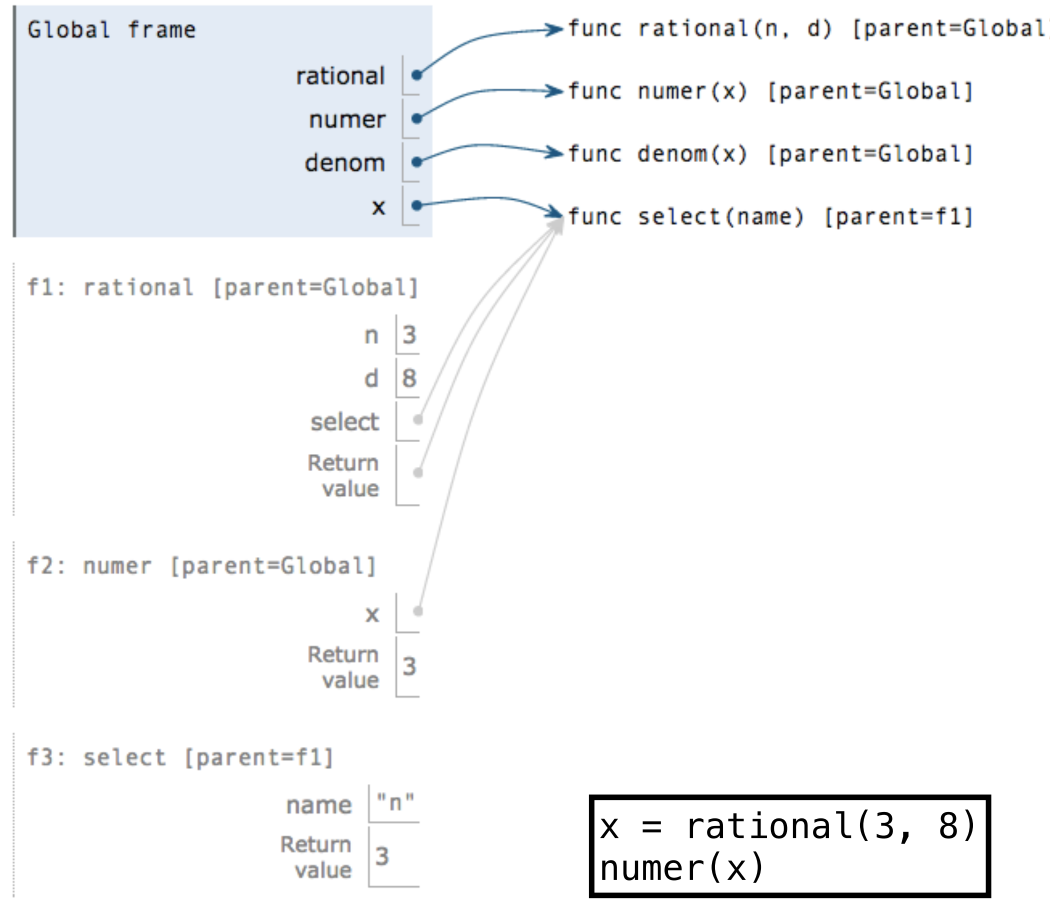
This function represents a rational number

Constructor is a higher-order function

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

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

Selector calls x




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

pythontutor.com/composingprograms.html#code=def%20rational%28n,%20d%29%3A%0A%20%20%20%20def%20select%28name%29%3A%0A%20%20%20%20%20%20%20%20if%20name%20%3D%3D%20'n'%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20n%0A%20%20%20%20%20%20%20%20elif%20name%20%3D%3D%20'd'%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20d%0A%20%20%20%20return%20select%0A%20%20%20%20%0Adef%20numer%28x%29%3A%0A%20%20%20%20return%20x%28'n'%29%0A%0Adef%20denom%28x%29%3A%0A%20%20%20%20return%20x%28'd'%29%0A%20%20%20%20%0Ax%20%3D%20rational%283,%208%29%0Anumer%28x%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=0

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

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

```
Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

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 *result dictionary* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   - A. Bind <name> to that element in the new frame from step 1
   - B. If <filter exp> evaluates to a true value, then add to the result dictionary an entry that pairs the value of <key exp> to the value of <value exp>

`{x * x: x for x in [1, 2, 3, 4, 5] if x > 2}` 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.

```
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}
```