

Lecture11 Data Abstraction

Data Abstraction

Data Abstraction



- Compound objects combine 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*



7

Rational Numbers



$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation may be 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

8

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

9

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)
```

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

```
def equal_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

10

Pairs

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

```
>>> x, y = pair
>>> x
```

```
1
>>> y
2
#this is called unpacking the list

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

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

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

13

Reducing to Lowest Terms



Example:

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

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

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd
```

Greatest common divisor

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

14

Abstraction Barriers

don't need to let the function know every part of the program, but only a part of it

moreover, it allows changes which would be taken advantage of by other parts and keep consistency

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

should not be using the function in another different "layer", obey the abstractions

? need a deeper understanding

use the level of abstraction appropriately

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 Representation

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 $\text{numer}(x)/\text{denom}(x)$ 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 data abstraction by its behavior

this is exactly the same to what happens in math! we don't actually care about how we define sth. in fact, we would say that, no matter what that have these characteristics we would call them the same thing! that's abstraction!

former

A screenshot of a Mac OS X terminal window titled "lec - bash - 52x28". The window shows a Python script named "ex.py" with code for rational arithmetic. The code defines several functions: add_rational, mul_rational, rationals_are_equal, print_rational, rational, numer, and denom. The man from the previous slide is visible in the top right corner of the terminal window.

```
# Rational arithmetic
def add_rational(x, y):
    """Add rational numbers x and y."""
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)

def mul_rational(x, y):
    """Multiply rational numbers x and y."""
    return rational(numer(x) * numer(y), denom(x) * denom(y))

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

def print_rational(x):
    """Print rational x."""
    print(numer(x), "/", denom(x))

# Constructor and selectors

def rational(n, d):
    """Construct a rational number x 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]
```

after

Constructor and selectors

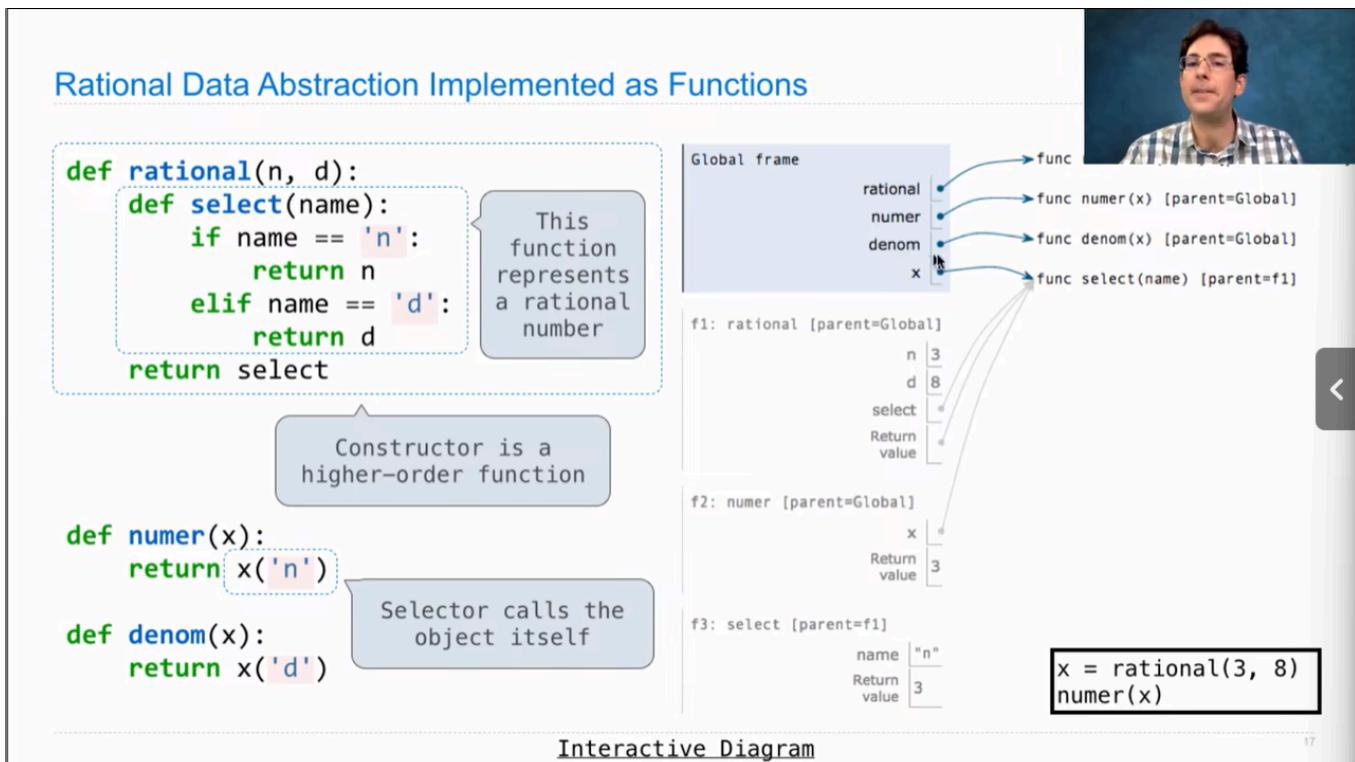
```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def numer(x):
    """Return the numerator of rational number x."""
    return x('n')

def denom(x):
    """Return the denominator of rational number x."""
    return x('d')
```

~
-- VISUAL --

we want to show the power of data abstraction here, where we change the representation but functions in different layers worked well.



Interactive Diagram

17

Dictionaries

```
Terminal Shell Edit View Window Help lec — Python — 104x28 41%
>>> {'I': 1, 'V': 5, 'X': 10}
{'X': 10, 'V': 5, 'I': 1}
>>> numerals = {'I': 1, 'V': 5, 'X': 10}
>>> numerals
{'X': 10, 'V': 5, 'I': 1}
>>> numerals['X']
10
>>> numerals[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 10
>>> numerals.keys()
dict_keys(['X', 'V', 'I'])
>>> numerals.values()
dict_values([10, 5, 1])
>>> numerals.items()
dict_items([('X', 10), ('V', 5), ('I', 1)])
>>> items = [('X', 10), ('V', 5), ('I', 1)]
>>> items
[('X', 10), ('V', 5), ('I', 1)]
>>> dict(items)
{'X': 10, 'V': 5, 'I': 1}
>>> dict(items)['X']
```

```
Terminal Shell Edit View Window Help lec — Python — 104x28 38%
>>> 'X' in numerals
True
>>> 'X-ray' in numerals
False
>>> numerals.get('X', 0)
10
>>> numerals.get('X-ray', 0)
0
>>> {x:x**x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> squares = {x:x**x for x in range(10)}
>>> squares[7]
49
>>> {1: 2, 1: 3}
{1: 3}
>>> {1: [2, 3]}
{1: [2, 3]}
>>> {[1]: 2}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```



Limitations on Dictionaries

Dictionaries are **unordered** 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