CS61A Lecture 14

Amir Kamil
UC Berkeley
February 22, 2013
The 61A Graffiti Bandit Strikes Again!

Thanks to Colin Lockard for the picture (and the title)!
Announcements

☐ HW5 out

☐ Hog contest due today
  ☐ Completely optional, opportunity for extra credit
  ☐ See website for details

☐ Trends project out today
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

Wishful thinking
Tuples

>>> 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 on tuples today
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:

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

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

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

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

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)
Abstraction Barriers

Rational numbers as whole data values

\[\text{add\_rational} \quad \text{mul\_rational} \quad \text{eq\_rational}\]

Rational numbers as numerators & denominators

\[\text{rational} \quad \text{numer} \quad \text{denom}\]

Rational numbers as tuples

\[\text{tuple} \quad \text{getitem}\]

However tuples are implemented in Python
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])
What is an Abstract Data Type?

- 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 $\frac{\text{numer}(x)}{\text{denom}(x)}$ must equal $\frac{n}{d}$.

- 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.
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

- \( \text{getitem}_\text{pair}(p, 0) \) returns \( x \), and
- \( \text{getitem}_\text{pair}(p, 1) \) returns \( y \).

Together, selectors are the inverse of the constructor.

Generally true of container types. \( \text{Not true for rational numbers because of GCD} \)
def pair(x, y):
    """Return a functional pair.""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

def getitem_pair(p, i):
    """Return the element at index i of pair p.""
    return p(i)

This function represents a pair

Constructor is a higher-order function

Selector defers to the functional pair
Using a Functionally Implemented Pair

```python
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
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!
```

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions.
The Sequence Abstraction

red, orange, yellow, green, blue, indigo, violet.

0, 1, 2, 3, 4, 5, 6.

There isn't just one sequence type (in Python or in general)

This abstraction is a collection of behaviors:

**Length.** A sequence has a finite length.

**Element selection.** A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

The sequence abstraction is shared among several types, including tuples.
Tuples introduce new memory locations outside of a frame

We use *box-and-pointer* notation to represent a tuple

- Tuple itself represented by a set of boxes that hold values
- Tuple value represented by a pointer to that set of boxes

Example: http://goo.gl/iFHx0
A method for combining data values satisfies the closure property if:

The result of combination can itself be combined using the same method.

Closure is the key to power in any means of combination because it permits us to create hierarchical structures.

Hierarchical structures are made up of parts, which themselves are made up of parts, and so on.

**Tuples can contain tuples as elements**
Recursive Lists

Constructor:

```python
def rlist(first, rest):
    """Return a recursive list from its first element and the rest."""
```

Selectors:

```python
def first(s):
    """Return the first element of recursive list s."""

def rest(s):
    """Return the remaining elements of recursive list s."""
```

Behavior condition(s):

If a recursive list \( s \) is constructed from a first element \( f \) and a recursive list \( r \), then

- \( \text{first}(s) \) returns \( f \), and
- \( \text{rest}(s) \) returns \( r \), which is a recursive list.
Implementing Recursive Lists Using Pairs

A recursive list is a pair

The first element of the pair is the first element of the list

The second element of the pair is the rest of the list

None represents the empty list

Example: http://goo.gl/fVhbF
Implementing the Sequence Abstraction

```python
def len_rlist(s):
    """Return the length of recursive list s."""
    if s == empty_rlist:
        return 0
    return 1 + len_rlist(rest(s))

def getitem_rlist(s, i):
    """Return the element at index i of recursive list s."""
    if i == 0:
        return first(s)
    return getitem_rlist(rest(s), i - 1)
```

**Length.** A sequence has a finite length.

**Element selection.** A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.