Announcements

- HWS out

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

- Trends project out today

Rational Number Arithmetic Code

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

Tuples

```python
>>> pair = (1, 2)
>>> x, y = pair
>>> x 1
>>> y 2
>>> pair[0] 1
>>> pair[1] 2
```

More on tuples today

Representing Rational Numbers

```python
def rational(n, d):
    """Construct a rational number x that represents n/d.""
    return (n, d)

from operator import getitem

def numerator(x):
    """Return the numerator of rational number x.""
    return getitem(x, 0)

def denominator(x):
    """Return the denominator of rational number x.""
    return getitem(x, 1)
```
Reducing to Lowest Terms

```python
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)
```

**Example:**

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \\
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

Abstraction Barriers

Rational numbers as whole data values

---

add_rational mul_rational eq_rational

---

Rational numbers as numerators & denominators

```
rational numer denom
```

Rational numbers as tuples

```
tuple getitem
```

However tuples are implemented in Python

Violating Abstraction Barriers

```python
add_rational( (1, 2), (1, 4) )
```

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

What is a pair?

- Does not use constructors
- Twice!

- No selectors!
- And no constructor!

Behavior Conditions of a Pair

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
  - \(*getitem_pair(p, 0)\) returns \( x \), and
  - \(*getitem_pair(p, 1)\) returns \( y \).

Together, selectors are the inverse of the constructor

Generally true of container types.

Not true for rational numbers because of GCD

Functional Pair Implementation

```python
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
        return dispatch
    return dispatch
```

This function represents a pair

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

Selector defers to the functional pair

Constructor is a higher-order function

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 \( \text{numerator}(x)/\text{denominator}(x) \) must equal \( n/d \).
- 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
Using a Functionally Implemented Pair

```python
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions.
```

This pair representation is valid!

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 in Environment Diagrams

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: [link]

The Closure Property of Data Types

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

- `first(s)` returns `f`, and
- `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: [link]
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.