Lecture #7: Data Abstraction

- Reprise: An Abstract Data Type (or ADT) consists of
  - A set (domain) of possible values.
  - A set of operations on those values.
- ADTs are conceptual: a given programming language may or may not have constructs specifically designed for ADT definition, but programmers can choose to organize their programs as collections of ADTs in any case.
- We call them "abstract" because they abstract a particular behavior, which we document without being specific about what the values really consist of (their internal representations).

Data Structures

- The simplest ADTs are not particularly abstract: they are a collection of data values and their behavior consists entirely of selecting or modifying those individual data values.
- We sometimes use the term data structure for these, although the terminology is not particularly exact.
- Example: A tuple is a sequence of values. It is entirely defined by those values.

Tuples in Python (I)

- To create construct a tuple, use a sequence of expressions in parentheses:
  - () # The tuple with no values
  - (1, 2) # A pair: tuple with two items
  - (1, ) # A singleton tuple: use comma to distinguish from (1)
  - (1, "Hello", (3, 4)) # Any mix of values possible.
- When unambiguous, the parentheses are unnecessary:
  - x = 1, 2, 3 # Same as x = (1,2,3)
  - return True, 5 # Same as return (True, 5)
  - for i in 1, 2, 3: # Same as for i in (1,2,3):

Tuples in Python (II)

- Basically, one can select values from a tuple and compare or print them, but little else.
- Select by item number:
  - x = (1, 7, 5)
  - print(x[1], x[2]) # Prints 7 and 5
  - from operator import getitem
  - print(getitem(x, 1), getitem(x, 2)) # Prints 7 and 5
  - print(x.__getitem__(1), x.__getitem__(2)) # Prints 7 and 5
- Or select by "unpacking" (syntactic sugar):
  - x = (1, (2, 3), 5)
  - a, b, c = x
  - print(b, c) # Prints (2, 3) and 5
  - d, (e, f), g = x
  - print(e, g) # Prints 2 and 5
- Tuples provide a useful way to return multiple things from a function.

Rational Numbers

- The book uses "rational number" as an example of an ADT:
  - def make_rat(n, d):
  -   """The rational number n/d, assuming n, d are integers, d!=0""
  - def add_rat(x, y):
  -   """The sum of rational numbers x and y.""
  - def mul_rat(x, y):
  -   """The product of rational numbers x and y.""
  - def numer(r):
  -   """The numerator of rational number r.""
  - def denom(r):
  -   """The denominator of rational number r.""
- These definitions pretend that x, y, and r really are rational numbers.
- But from this point of view, numer and denom are problematic. Why?
Representing Rationals (I)

- The obvious representation is as a pair of integers.
- Suppose we define

```python
def make_rat(n, d):
    """Rational number n/d, assuming n, d are integers, d!=0""
    return (n, d)
```

- From elementary-school math, we can then write

```python
def add_rat(x, y):
    """The sum of rational numbers x and y.""
    (xn, xd), (yn, yd) = x, y
    return (xn * yd + yn * xd, xd * yd)  # BAD STYLE?

def mul_rat(x, y):
    """The product of rational numbers x and y.""
    (xn, xd), (yn, yd) = x, y
    return (xn * yn, xd * yd)  # BAD STYLE?
```

- What about `numer` and `denom`?

Use the Abstraction!

Better:

```python
def add_rat(x, y):
    """The sum of rational numbers x and y.""
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                    denom(x) * denom(y))

def mul_rat(x, y):
    """The product of rational numbers x and y.""
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

Implementing numer and denom (I)

```python
from fractions import gcd

# fractions.gcd(a,b), for b!=0, computes the largest integer in
# absolute value that evenly divides both a and b and has
# the sign of b. (Not quite the "official" gcd function).

def numer(r):
    """The numerator of rational number r in lowest terms.""
    n, d = r
    return n // gcd(n, d)

def denom(r):
    """The denominator of rational number r in lowest terms. Always positive.""
    n, d = r
    return d // gcd(n, d)
```

Representing Rationals (II)

- But the preceding implementation is problematic:
  - Each call to `denom` or `numer` has to recompute a value.
  - Intermediate values can get quite large.
- Suggests that we always keep rationals in lowest terms.
- How does the implementation change?

Updated Implementation

```python
from fractions import gcd

def make_rat(n, d):
    g = gcd(n, d)
    return n//g, d//g

def numer(r):
    return r[0]

def denom(r):
    return r[1]

# What about add_rat and mul_rat?
```

Updated Implementation (contd.)

```python
def add_rat(x, y):
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                    denom(x) * denom(y))

def mul_rat(x, y):
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```
Implementing Tuples (If You Had To)

- Using "data structure" to mean "unabstract ADT" is fuzzy.
- Even tuples need to be represented.
- Python has a built-in implementation, inaccessible to the user.
- They do this for speed, but we can get the same effect with what we already have: functions.

```python
def make_rat(n, d):
    g = gcd(n, d)
    n, d = n // g, d // g
    def result(key):
        if key == 0:
            return n
        else:
            return d
    return result

def numer(r):
    return r(0)

def denom(r):
    return r(1)
```

- The function `result` dispatches on the value of `key` to get its result.

Discussion

- You'll sometimes see `key` described as a `message` and this technique called `message-passing`, (but your current instructor hates this terminology.)
- If we had persisted in defining `add_rat` and `mul_rat` using unpacking, as originally (see slide 7), we'd now have to rewrite them.
- But by using `numer` and `denom` in `add_rat` and `mul_rat` (slide 8), we have avoided having to touch them after this change in representation.
- The general lesson:
  
  *Try to confine each design decision in your program to as few places as possible.*