Lecture #10: Abstractions: From Function to Data

Announcements:

- Watch Piazza, home page for news concerning review on Monday.
- If you haven't responded to the Welcome Survey in HW#1, please do so. We're about 200 responses shy.
- Quiz results. Out of 3 questions: 18% got 3, 46% got 2, 36% got 1, and 9% got 0.
- Please talk to your TA if you got 0 or did not turn in the quiz (or get a response).
- Project due Thursday (13 Feb) at midnight (11:59+).
- Test #1 Tuesday night 8–10PM in rooms to be announced (watch Piazza).
- DSP students: You'll get mail about an alternative location. Your test will overlap the main test time.
- Alternative test time: Wednesday morning at 9AM (TBA). Please see us if you can't make that time.

Separation of Concerns

- The sierpinski routine used triangle.
- To write sierpinski, I needed only to know:
  - The *syntactic specification* of triangle: its name and number of arguments (given by its **def** header), and
  - Its *semantic specification*: what a call does or means (given by its documentation comment).
- I did not need to know how triangle works or who else calls it.
- Likewise, triangle does not need to know
  - where its arguments come from,
  - who calls it, or
  - what use is made of its return value or side effects.
- There is a *separation of concerns* between these functions.
- This is a fundamental concept in software engineering: organize programs so that you can work on one thing at a time in isolation.

Names

Semantically, names are arbitrary; to the reader, they are part of the documentation.

Bad:	**Better:**
number	dice_rolls
true_false	pigged_out
d	dice, die
helper	take_turn, find_repeat
do_stuff	rescale_figure
random obscenity	report_error
l, I, O	k, m, n

Names convey meaning or purpose to the programmer (not to the machine).

Function names should convey their value (abs, sqrt) or effect (print)

Use the documentation comments of functions to elaborate where necessary, to indicate the types of arguments and return values, and to indicate assumptions or limitations on the arguments.

Function Comments

Comments on a function should suffice to tell the reader everything needed to use it.

Rather than

```
def largest(L):
    """Find the largest value"""
    k = 0
    for i in range(1, len(L)):
        if L[i] > L[k]:
            k = i
    return k
```

Use

```
def largest(L):
    """Return the index of the largest
    value in L."""
    k = 0
    for i in range(1, len(L)):
        if L[i] > L[k]:
            k = i
    return k
```

Names and Comments

- I generally limit comments to
  - Docstrings on functions (or later, on classes)
  - Comments and documentation at the beginning of a module describing its purpose, conventions, authorship, copyright permissions, etc.
  - Comment names of significant constants.
- Avoid internal comments: they indicate places where you could make a function shorter or use a better name:

Rather than	*Use*
`# Compute the discriminant` `d = b**2 - 4*a*c`	`discriminant = b**2 - 4*a*c`

Refactoring

- Your comments can suggest to you that things are getting too big, or that a function is doing to much.
- When that happens, it is time to *refactor:* break functions up into more coherent pieces.
- Consider the function:

```
def print_averages(grade_book, out):
    """Compute the average scores for each student in
    GRADE_BOOK and prints on OUT."""
```

- What if we just want to know the averages?
- What if we also want a different format, including other information?
- Makes more sense, e.g., to have a get_averages function, and a more general print routine that will print any information about students.

Unit Testing

- The docstring tests that you execute with `python3 -m doctest` are examples of *unit tests.*
- That is, tests on the smallest testable units of your program (functions).
- *Test-driven development* refers to the practice of creating tests *ahead of* implementation.
- Don't wait for your program to be finished to test it.
- The doctest Python module makes it possible to run all your tests cumulatively, watching for inadvertant errors and tracking how much still needs to be done.

Decorators

- You've seen functions on functions. They can also be used for testing or debugging:

```
def trace1(fn):
    """Return a function equivalent to FN, a one-argument
    function, that also prints trace output."""

    def traced(x):
        print('Calling', fn, 'on argument', x)
        return fn(x)
    return traced
```

- To use this:

```
def triple(x):
    return 3*x
triple = trace1(triple)
```

- Or, more conveniently, do the equivalent with Python's decorators:

```
@trace1
def triple(x):
    return 3*x
```

Abstract Data Types

- 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 exactly firm.
- Example: A *tuple* is a sequence of values. It is entirely defined by those values.

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?

Rational Numbers

- Problem is that "the numerator (denominator) of r" is not well-defined for a rational number.
- If make_rat really produced rational numbers, then make_rat(2, 4) and make_rat(1, 2) ought to be identical. So should make_rat(1, -1) and make_rat(-1, 1).
- So a better specification would be

```
def numer(r):
    """The numerator of rational number R in lowest terms."""

def denom(r):
    """The denominator of rational number R in lowest terms.
    Always positive."""
```

Representing Rationals (I)

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

```
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

```
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:

```
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)

```
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

```
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 happens to add_rat and mul_rat?
- **Ans:**

Updated Implementation

```
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 happens to add_rat and mul_rat?
- **Ans:** They do not change! The use of the make_rat abstraction makes it unnecessary.

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.

Data Structures via Dispatching

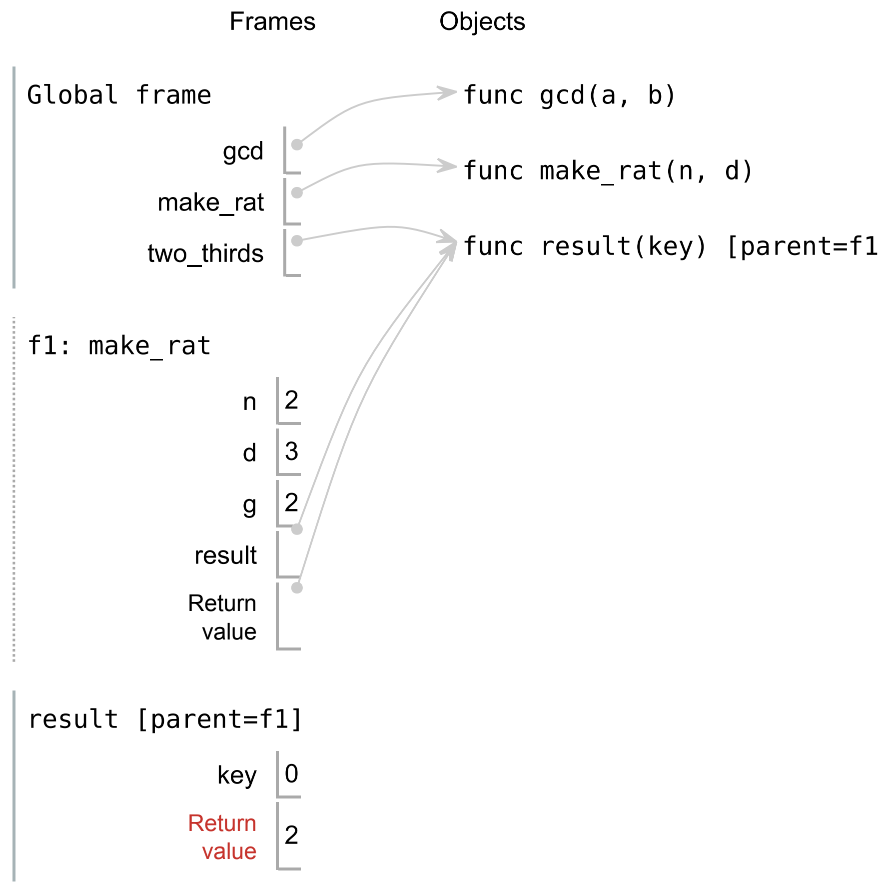
```
def make_rat(n, d):
    """A function, r, representing the rational number N/D.
    r(0) is the numerator and r(1)>0 the denominator (in lowest
    terms)."""

    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)
```

- We say that the function result *dispatches* on the value of key.
- The tuple in the previous representation is now replaced by the *environment frame* created by a call to make_rat.

```python
def gcd(a, b):
    a, b, s = min(abs(a), abs(b)), max(abs(a), abs(b)),
    while a != 0: a, b = b%a, a
    return b

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

two_thirds = make_rat(4, 6)
two_thirds(0)
```

Edit code

<< First | < Back | Step 21 of 21 | Forward > | Last >>



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.