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.

<table>
<thead>
<tr>
<th>Bad:</th>
<th>Better:</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>dice_rolls</td>
</tr>
<tr>
<td>true, false</td>
<td>pigged, out</td>
</tr>
<tr>
<td>d</td>
<td>dice, die</td>
</tr>
<tr>
<td>helper</td>
<td>take_turn,</td>
</tr>
<tr>
<td>do_stuff</td>
<td>rescale, figure</td>
</tr>
<tr>
<td>random</td>
<td>report_error</td>
</tr>
<tr>
<td>obscenity</td>
<td>k, m, n</td>
</tr>
</tbody>
</table>

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

**Rather than**

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

**Use**

```python
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:

  ```python
  # Compute the discriminant
  discriminant = b**2 - 4*a*c
  d = b**2 - 4*a*c
  ```

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

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

  ```python
  def print_averages(grade_book, 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 inadvertent 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:
  ```python
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:
  ```python
def triple(x):
    return 3*x
triple = trace1(triple)
```
- Or, more conveniently, do the equivalent with Python’s decorators:
  ```python
@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:
  ```python
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))
  ```

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 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.

Discussion

• You'll sometimes see key described as a message and this technique
called message-passing, (but your current instructor hates this ter-
minalogy.)
• 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 representa-
tion.
• The general lesson:
  
  Try to confine each design decision in your program
to as few places as possible.