Lecture #9: Abstractions and Objects
Data Abstraction

• Functions are abstractions that represent computations and actions.
• In the old days, one described programs as hierarchies of actions: *procedural decomposition*.
• Starting in the 1970’s, emphasis moved to the data that the functions operate on.
• An *abstract data type (ADT)* represents some kind of thing and the operations upon it.
• Instances of the type are often generically called *objects*.
• We can usefully organize our programs around the abstract data types in them.
• For each type, we define an *interface* that describes for users (“clients”) of that type of data what operations are available.
• Typically, the interface consists of functions.
• The collection of specifications (syntactic and semantic—see lecture #6) constitute a specification of the type.
• We call ADTs *abstract* because clients ideally need not know internals.
Functions as Data Abstractions (I)

- Functions can serve as objects.
- In the path-finding example, the `blocked` argument was a function.
- But it essentially represented data: the set of places that were blocked.
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?
A Better Specification

• 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

  ```python
  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."""
  ```
We have a tool that can implement this specification now: functions.

```python
from math import gcd  # Need Python3.5 actually.

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return lambda which: n if which == 0 else d

def numer(r):
    """The numerator of rational number r.""
    return ?

def denom(r):
    """The denominator of rational number r.""
    return ?
```
Representation as Functions (II)

- Rational numbers represented as functions that take a single argument and reveal one of their local variables.

```python
from math import gcd

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return lambda flag: n if flag == 0 else d

def numer(r):
    """The numerator of rational number r.""
    return r(0)

def denom(r):
    """The denominator of rational number r.""
    return r(1)

def add_rat(x, y):
    """The sum of rational numbers x and y.""
    return ?

def mul_rat(x, y):
    """The product of rational numbers x and y.""
    return ?
```
Representation as Functions (III)

- One possibility:

```python
from math import gcd

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return lambda flag: n if flag == 0 else d

... def add_rat(x, y):
    n0, n1, d0, d1 = x(0), y(0), x(1), y(1)
    n, d = n0 * d1 + n1 * d0, d0 * d1
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return lambda flag: n if flag == 0 else d

    def mul_rat(x, y):
        """The product of rational numbers x and y."""
        return ?
```

- Comments?
Abstraction Violations and DRY

• Having created an abstraction (make_rat, numer, denom), use it:
  - Then, later changes of representation will affect less code.
  - Code will be clearer, since well-chosen names in the API make intent clear.

... 

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

Last modified: Mon Feb 22 16:35:21 2016
Objects in Python

• In Python 3, every value is a reference to an object.
• Varieties of object correspond (roughly) to classes (types).
• Each object has some set of attributes, accessible using dot notation, which are values:
  - E.Attr, where E is a simple expression and Attr is a name, means “the current value of the Attr attribute of the object referred to by the value of E.”
• Among these attributes are those whose values are a kind of function known as a method.
• For historical reasons or notational clarity, there are often alternative ways to access attributes than dot notation:
  - x.__add__(y) | add(x, y) or x+y
  - L.__getitem__(k) | L[k]
  - x.__len__() | len(x)
  - x.__eq__(y) | x == y
Primitive Types: Numbers

- A *primitive type* is one that is built into a language, possibly with characteristics or syntax that cannot be written into user-defined types.

- In Python, numbers are such types: have their own literals and internal attributes that are not accessible to the programmer.

- Python distinguishes four types:
  - `int`: Integers.
  - `bool`: Limited integers restricted to two values equivalent to 0 and 1: *False* and *True*.
  - `float`: A subset of the rational numbers used to approximate real-valued quantities.
  - `complex`: A subset of the rational complex numbers used to approximate complex-valued quantities.
Primitive Types: Tuples

- **tuple** is another primitive type with special syntax.
- To create *construct* a tuple, use a sequence of expressions in parentheses:

  ```python
  ()       # 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:

  ```python
  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:**

```python
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):**

```python
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
```

• **A useful way to return multiple things from a function.**
A (More Conventional) Representation of Rational

- Yes, we *can* use functions to represent data, but it's not common practice.
- Quite a bit of overhead, both in space and time.
- So, let's use tuples instead:

```python
from math import gcd

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    return (n//g, d//g)

def numer(r):
    """The numerator of rational number r.""
    return r[0]

def denom(r):
    """The denominator of rational number r.""
    return r[1]
```

- What else changes *(add_rat, mul_rat)*?
Discussion

• 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:

   \textit{Try to confine each design decision in your program to as few places as possible.}
math.gcd was introduced into Python in version 3.5. In its absence, you can implement it yourself:

```python
def gcd(a, b):
    if a == b == 0:
        return 1
    a, b = abs(a), abs(b)
    if a > b:
        a, b = b, a
    while a != 0 != b:
        a, b = b % a, a
    return b
```