DATA ABSTRACTION 4

COMPUTER SCIENCE 61A

September 25th, 2013

1 Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects — for example, car objects, chair objects, people objects, etc. That way, programmers don't have to worry about *how* code is implemented — they just have to know *what* it does.

This is especially important when programming with other people: with data abstraction, your group members won't have to read through every line of your code to understand how it works before they use it — they can just assume that it does work.

Data abstraction mimics how we think about the world. For example, when you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

To facilitate data abstraction, you will need to create two types of functions: constructors and selectors. Constructors are functions that build the abstract data type. Selectors are functions that retrieve information from the data type.

For example, say we have an abstract data type called `city`. This `city` object will hold the `city`'s name, and its latitude and longitude. To create a `city` object, you'd use a function like

```
city = make_city(name, lat, lon)
```

To extract the information of a `city` object, you would use functions like

```
get_name(city)
get_lat(city)
get_lon(city)
```

The following code will compute the distance between two city objects:

```
from math import sqrt
def distance(city1, city2):

    lat_1, lon_1 = get_lat(city_1), get_lon(city_1)
    lat_2, lon_2 = get_lat(city_2), get_lon(city_2)

    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Notice that we don't need to know how these functions were implemented. We are assuming that someone else has defined them for us.

It's okay if the end user doesn't know how functions were implemented. However, the functions still have to be defined by someone. We'll look into defining the constructors and selectors later in this discussion.

1.1 Data Abstraction Practice

1. Implement closer_city, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is closer.

   You may only use selectors and constructors (introduced above) for this question. You may also use the distance function defined above.

   ```
   def closer_city(lat, lon, city1, city2):
   ```

   Solution:

   ```
       new_city = make_city('arb', lat, lon)
       dist1 = distance(city1, new_city)
       dist2 = distance(city2, new_city)
       if dist1 < dist2:
           return get_name(city1)
       return get_name(city2)
   ```

2 Let's be rational!

In lecture, we discussed the rational data type, which represents fractions with the following methods:

`rational(n, d)` - constructs a rational number with numerator `n`, denominator `d`
`numer(x)` - returns the numerator of rational number `x`
`denom(x)` - returns the denominator of rational number `x`

We also presented the following methods that perform operations with rational numbers:

```
add_rationals(x, y)
mul_rationals(x, y)
eq_rationals(x, y)
```

You'll soon see that we can do a lot with just these simple methods.

2.1 Rational Number Practice

1. Write a number that returns the given rational number `x` raised to positive power `e`.

   ```
   from math import pow
   def rational_pow(x, e):
   ```

   Solution:

   ```
       return rational(pow(numer(x), e), pow(denom(x), e))
   ```

2. The irrational number $e \approx 2.718$ can be generated from an infinite series. Let's try calculating it using our rational number data type! The mathematical formula is as follows:

   $$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \cdots$$

   Write a function `approx_e` that returns a rational number approximation of e to `iter` amount of iterations. We've provided a factorial function.

   ```
   def factorial(n):
       return 1 if n == 0 else n * factorial(n - 1)
   def approx_e(iter=100):
   ```

   Solution:

   ```
       k = 0
       e = rational(0, 1)
       while k < iter:
   ```

```
        e = add_rationals(e, rational(1, factorial(k)))
        k += 1
    return e
```

3. Implement the following rational number methods.

```
def inverse_rational(x):
    """Returns the inverse of the given non-zero rational number"""
```

Solution:

```
    return rational(denom(x), numer(x))
```

```
def div_rationals(x, y):
    """Returns x / y for given rational x and non-zero rational y"""
```

Solution:

```
    return mul_rationals(x, inverse_rational(y))
```

3 My Life for Abstraction

We've used data abstractions up to this point. Now let's try creating some ourselves. So far, we know of two ways of creating the pair abstraction.

3.1 Tuples, or, Zerg Rush!

Remember, a pair is a *compound* data type that holds two other pieces of data. So far, we have provided you with two ways of representing the pair data type. The first way to implement pairs is with the Python tuple construct.

```
>>> nums = (1, 2)
>>> nums[0]
1
>>> nums[1]
2
```

Note how we use the square bracket notation to access the data we stored in the pair. The data is *zero indexed*, meaning we access the first element with `nums[0]` and the second with `nums[1]`.

Let’s now use data abstractions to recreate the popular video game Starcraft: Brood War. In Starcraft, the three races, Zerg, Protoss, and Terran, create ”units” that they send to attack each other.

1. Implement the constructors and selectors for the unit data abstraction using tuples. Each unit will have a string catchphrase and an integer amount of damage.

```
def make_unit(catchphrase, damage):
```

Solution:

```
    return (catchphrase, damage)
```

```
def get_catchphrase(unit):
```

Solution:

```
    return unit[0]
```

```
def get_damage(unit):
```

Solution:

```
    return unit[1]
```

3.2 Data Abstraction Violations, or, I Long For Combat!

Data abstraction violations happen when we assume we know something about how our data is represented. For example, if we use pairs and we forget to use a selector and instead use the index.

```
>>> raynor = make_unit('This is Jimmy.', 18)
>>> print(raynor[0]) # violation!!!!
This is Jimmy.
```

In this example, we assume that `raynor` is represented as a tuple because we use the square bracket indexing. However, we should have used the selector `get_catchphrase`. This is a data abstraction violation.

2. Units attack each other in events called battles. Let’s simulate these battles. In a battle, each unit yells its respective catchphrase, then the unit with more damage wins the

battle. Implement `battle`, in which the you print the catchphrases of the first and second in that order, then return the unit that does more damage. The first unit wins ties. Don't violate any data abstractions!

```
def battle(first, second):
    """Simulates a battle between the first and second unit
    >>> zealot = make_unit('My life for Aiur!', 16)
    >>> zergling = make_unit('GRAAHHH!', 5)
    >>> winner = battle(zergling, zealot)
    GRAAHHH!
    My life for Aiur!
    >>> winner is zealot
    True
    """
```

Solution:

```
    print(get_catchphrase(first))
    print(get_catchphrase(second))
    if get_damage(first) >= get_damage(second):
        return first
    return second
```

3.3 Functional Pairs, or, We Require More Minerals

The second way of constructing pairs is with higher order functions. We can implement the functions `pair` and `getitem_pair` to achieve the same goal.

```
>>> def pair(x, y):
        """Return a function that behaves like a two-element tuple"""
        def dispatch(m):
            if m == 0:
                return x
            elif m == 1:
                return y
        return dispatch
>>> def getitem_pair(p, i):
        """Return the element at index i of pair p"""
        return p(i)
>>> nums = pair(1, 2)
>>> getitem_pair(nums, 0)
1
>>> getitem_pair(nums, 1)
2
```

Note how although using functional pairs is different syntactically from tuples, it accomplishes the exact same thing.

Continuing with our Starcraft example, we now want to structure the way in which we create units. Units require resources to create, and in Starcraft, these resources are called ”minerals” and ”gas.”

1. Write constructors and selectors for a data abstraction that combines an integer amount of minerals and gas together into a bundle. Use functional pairs.

```
def make_resource_bundle(minerals, gas):
```

Solution:

```
    return pair(minerals, gas)
```

```
def get_minerals(bundle):
```

Solution:

```
    return getitem_pair(bundle, 0)
```

```
def get_gas(bundle):
```

Solution:

```
    return getitem_pair(bundle, 1)
```

3.4 Putting It All Together

The beauty of data abstraction is that we can treat complex data in a very simple way. Although we’ve only been dealing with storing primitive data types inside our pairs, we can in fact store more complex data in the exact same way. A simple example is nesting tuples inside each other.

```
>>> def make_pair(a, b):
        return (a, b)
>>> def get_pair(pair, i):
        return pair[i]
>>> def make_pair_of_pairs(pair1, pair2):
        return make_pair(pair1, pair2)
```

```
>>> p = make_pair_of_pairs(make_pair(1, 2), make_pair(3, 4))
>>> get_pair(get_pair(p, 0), 0)
1
>>> get_pair(get_pair(p, 1), 0)
3
```

Let's apply this to our running Starcraft example. In Starcraft, buildings are used to create units, if given enough resources.

1. Let's make a `building` pair that is constructed with a unit data type and a resource bundle data type. This time take your choice of tuples or functional pairs in representing a building. Make sure not to violate any data abstractions.

   ```
   def make_building(unit, bundle):
   ```

   Solution:

   ```
       return (unit, bundle)
   ```

   or

   ```
       return pair(unit, bundle)
   ```

   ```
   def get_unit(building):
   ```

   Solution:

   ```
       return building[0]
   ```

   or

   ```
       return getitem_pair(building, 0)
   ```

   ```
   def get_bundle(building):
   ```

   Solution:

   ```
       return building[1]
   ```

   or

   ```
       return getitem_pair(building, 1)
   ```

   Your last task is to implement the `build_unit` method, that when given a building and sufficient amount of resources, builds a unit and returns it.

2. Implement `build_unit`. It is given a building and resource bundle. First, it checks whether the amount of resources provided is greater than or equal to the amount the building was constructed with. If it is not, it prints out some error message. Otherwise, it creates a copy of the building's unit and returns it.

```
def build_unit(building, bundle):
    """Constructs a unit if given the minimum amount of resources.
    Otherwise, prints an error message
    >>> barracks = make_building(make_unit('Go go go!', 6),
    ... make_resource_bundle(50, 0))
    >>> marine = build_unit(barracks, make_resource_bundle(20, 20))
    We require more minerals!
    >>> marine = build_unit(barracks, make_resource_bundle(50, 0))
    >>> print(get_catchphrase(marine))
    Go go go!
    """
```

Solution:

```
    if get_minerals(bundle) < get_minerals(get_bundle(building)):
        print("We require more minerals!")
    if get_gas(bundle) < get_gas(get_bundle(building)):
        print("We require more vespene gas!")
    else:
        unit = get_unit(building)
        return make_unit(get_catchphrase(unit), get_damage(unit))
```

3. Data abstractions are extremely useful when the underlying implementation of the abstraction changes. For example, after writing a program using tuples as a way of storing pairs, suddenly someone switches the implementation to functional pairs. If we correctly use constructors and selectors instead of assuming the pairs are tuples, our program should still work perfectly.

   Reimplement the `resource` abstraction to use tuples instead of functional pairs. Then verify that all the code that use the `resource` abstraction would still work.

```
def make_resource_bundle(minerals, gas):
```

Solution:

```
    return (minerals, gas)
```

```
def get_minerals(bundle):
```

Solution:

```
    return bundle[0]
```

```
def get_gas(bundle):
```

Solution:

```
    return bundle[1]
```