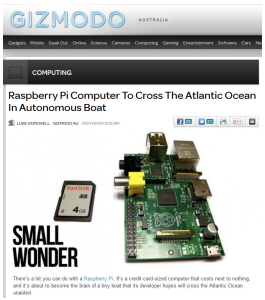
CS61A Lecture 8
Data Abstraction

Tom Magrino and Jon Kotker
UC Berkeley EECS

June 28, 2012

Computer Science in the News



Today

- Quick orders of growth review
- Data Abstraction and making new Abstract Data Types
- Tuples

Review: Orders of Growth

What is the order of growth (using Θ notation) for the following Python function?

```
def foo(x):
    if x < 3:
        return x
    return foo(x % 3) + foo(x - 1)
```


Review: Orders of Growth

Θ(n)

We know that the result of n % 3 is 0, 1, or 2 (the base case), so we know that the first recursive call will always result in a base case and we can treat it as a constant time operation.

The second recursive call will take (about) n recursive calls before reaching a base case (we subtract one from n each time). So we have Θ(n) recursive calls with constant amount of work done for each call.

Data Abstraction

We want to be able to think about data in terms of its *meaning* rather than in terms of the way it is represented.

Data abstraction allows us to isolate:

- How the data is *represented* (as parts)
- How the data is *manipulated* (as units)

We do this by using functions to help create a **division** between these two cases.

Problem: Rational Numbers

$$\frac{Numerator}{Denominator}$$

Exact representation of fractions using a pair of integers.

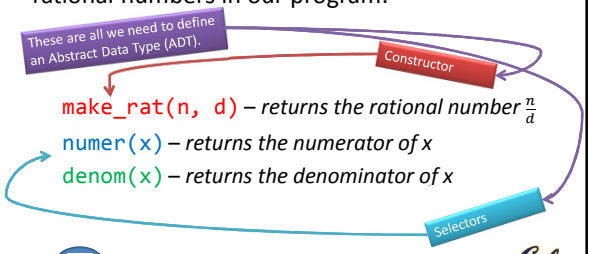
Multiplication $\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$

Addition $\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$

Equality $\frac{a}{b} = \frac{c}{d} \Leftrightarrow ad = cb$

Problem: Rational Numbers

We'd like to be able to create and decompose rational numbers in our program:



Problem: Rational Numbers

```
def mul_rats(r1, r2):
        return make_rat(numer(r1) * numer(r2), denom(r1) * denom(r2))

def add_rats(r1, r2):
        n1, d1 = numer(r1), denom(r1)
        n2, d2 = numer(r2), denom(r2)
        return make_rat(n1 * d2 + n2 * d1, d1 * d2)

def eq_rats(r1, r2):
        return numer(r1) * denom(r2) == numer(r2) * denom(r1)
```

Notice that we don't have to know how rational numbers work in order to write any code that uses them!

Problem: Rational Numbers

Great! If we can implement `make_rat`, `numer`, and `denom`, then we can finish our wonderful rational numbers module!

Practice: Using Abstractions

How would I write a function to invert (flip) a rational number using the constructor and selectors we are using for rational numbers?

Practice: Using Abstractions

How would I write a function to invert (flip) a rational number using the constructor and selectors we are using for rational numbers?

```
def invert_rat(r):
    return make_rat(denom(r), numer(r))
```

Tuples: Our First Data Structure

Tuples are a built-in datatype in Python for representing a **constant sequence** of data.

```
>>> pair = (1, 2)
>>> pair[0]
1
>>> pair[1]
2
>>> x, y = pair
>>> x
1
>>> y
2
>>> z = pair + (6, 5, 4)
>>> z
(1, 2, 6, 5, 4)
>>> len(z)
5
```

```
>>> z[2:5]
(6, 5, 4)
>>> triplet = (1,
...             2,
...             3)
>>> triplet
(1, 2, 3)
>>> for num in triplet:
...     print(num, "potato")
...
1 potato
2 potato
3 potato
>>> (1)
1
>>> (1,)
(1,)
```

Tuples: Our First Data Structure

The Python data type **tuple** is an example of what we call a *data structure* in computer science.

A *data structure* is a type of data that exists primarily to hold other pieces of data in a specific way.

Practice: Using Tuples and Abstractions

Write the higher order function `map`, which takes a function, `fn`, and a tuple of values, `vals`, and returns the tuple of results of applying `fn` to each value in `vals`.

```
>>> map(square, (1, 2, 3, 4, 5))
(1, 4, 9, 16, 25)
```

Practice: Using Tuples and Abstractions

Write the higher order function `map`, which takes a function, `fn`, and a tuple of values, `vals`, and returns a the tuple of results of applying `fn` to each value in `vals`.

```
def map(fn, vals):
    results = ()
    for v in vals:
        results = results + (fn(v),)
    return results
```

Announcements

- Project 1 autograder is running now.
- Next week, we will move to **105 Stanley** for the rest of the summer.
- Midterm 1 is on **July 9**.
  - We will have a review session closer to the date.
- If you need accommodations for the midterm, please notify DSP by the end of this week.
- HW1 grade should be available on `glookup`.

Problem: Rational Numbers

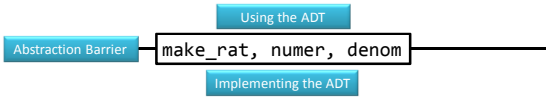
```
def make_rat(n, d):
    return (n, d)

def numer(x):
    return x[0]

def denom(x):
    return x[1]
```

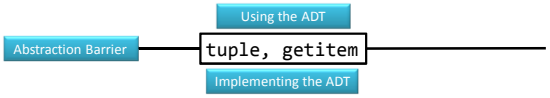
Abstraction Diagrams

rational numbers as numerators and denominators



rational numbers as tuples

tuples as sequences of data



However Python implements tuples

Data Abstraction: So What?

It makes code more readable and intuitive.

Which version is clearer?

```
def mul_rats(r1, r2):
    return make_rat(numer(r1) * numer(r2), denom(r1) * denom(r2))
```

```
def mul_rats(r1, r2):
    return (r1[0] * r2[0], r1[1] * r2[1])
```

When we write code that assumes a specific implementation of our ADT, we call this a ***data abstraction violation (DAV)***.

Data Abstraction: So What?

We don't have to worry about changing all the code that uses our ADT if we decide to change the implementation!

```
def make_rat(n, d):
    return (d, n)
def numer(x):
    return x[1]
def denom(x):
    return x[0]
```

```
# Will still work
def mul_rats(r1, r2):
    return make_rat(numer(r1) * numer(r2),
                    denom(r1) * denom(r2))
```

```
# Will break
def mul_rats(r1, r2):
    return (r1[0] * r2[0], r1[1] * r2[1])
```

Practice: Data Abstraction

Suppose that Louis Reasoner wrote the following function `prod_rats` that takes a tuple of rational numbers using our ADT and returns their product. Correct his code so that he does not have any data abstraction violations.

```
def prod_rats(rats):
    total, i = (1, 1), 0
    while i < len(rats):
        total = (total[0] * rats[i][0],
                 total[1] * rats[i][1])
        i += 1
    return total
```

Practice: Data Abstraction

Suppose that Louis Reasoner wrote the following function `prod_rats` which takes a tuple of rational numbers using our ADT and returns their product. Correct his code so that he does not have any data abstraction violations.

```
def prod_rats(rats):
    total, i = make_rat(1, 1), 0
    while i < len(rats):
        total = make_rat(numer(total) * numer(rats[i]),
                         denom(total) * denom(rats[i]))
        i += 1
    return total
```

Practice: Data Abstraction

Say I wrote the following functions to define my student ADT.

```
def make_student(name, id):
    return (name, id)
def student_name(s):
    return s[0]
def student_id(s):
    return s[1]
```

If I changed the student ADT to also include the student's age, what functions would I have to add or change in order to complete the abstraction?

Practice: Data Abstraction

Say I wrote the following functions to define my student ADT.

```
def make_student(name, id):
      return (name, id)
def student_name(s):
      return s[0]
def student_id(s):
      return s[1]
```

If I changed the student ADT to also include the student's age, what functions would I have to add or change in order to complete the abstraction?

You would have to change make_student to take this new parameter. If you just represent a student as the tuple (name, id, age), then you only have to add a selector for the student's age. The other two selectors would not have to be modified in this case.

Conclusion

- Tuples are a nice way to group data in Python.
- Learned how to design new types of data by using *data abstraction.*
- ***Preview***: Useful data structures.

Extras: Using Functions to Create ADTs

It turns out you don't need to have something like tuples in a language in order to group data together. Say I wanted to make a pair abstraction, which is like a tuple of length 2. I could do this with just functions:

```
def make_pair(first, second):
      def pair(msg):
            if msg == "first":
                  return first
            elif msg == "second":
                  return second
      return pair
def first(p):
      return p("first")
def second(p):
      return p("second")
```

Extras: More about Strings

Strings and tuples are both *sequences,* meaning that they are things that you can iterate over with a `for` loop. Interestingly, they can also be indexed into and sliced like tuples.

```
>>> for letter in "abc":
...     print(letter)
a
b
c
>>> "asdf"[2]
d
>>> "slaughterhouse"[1:9]
"laughter"
```