Databases and Algorithms

Abstract Data Types

Instructor: Rossano Schifanella

@SDS

Abstract Data Type (ADT)

- **Mathematical model** of the data objects that make up a data type, as well as the **functions** that operate on these objects (and logical or other relations between objects).
- ADT consist of two parts:
  - **data objects**
  - **operations with data objects**
- The term data type refers to the **implementation** of the mathematical model specified by an ADT

Abstract Data Type (ADT)

- The term **data structure** refers to a **collection of computer variables that are connected in some specific manner**
- The notion of data type includes basic data types. Basic data types are related to a programming language.
  - E.g., “int” in C/Java/Python

Python primitive types

- **Numeric types:**
  - **int**: Integers; equivalent to C longs in Python 2.x, non-limited length in Python 3.x
  - **long**: Long integers of non-limited length; exists only in Python 2.x
  - **float**: Floating-Point numbers, equivalent to C doubles
  - **complex**: Complex Numbers
- **Sequences:**
  - **str**: String; represented as a sequence of 8-bit characters in Python 2.x, but as a sequence of Unicode characters (in the range of U+0000 - U+10FFFF) in Python 3.x
  - **byte**: a sequence of integers in the range of 0-255; only available in Python 3.x

Example: Integer Set Data Type

- **Abstract Data Type**
  - Mathematical set of integers, I
  - Possible data items: -inf, …, -1, 0, 1, …, +inf
  - Operations: +, -, *, mod, div, etc.
- Actual, Implemented Data Type (available in Python):
  - It's the primitive type int
  - Only has range of -2^{31} to $2^{31} - 1$ for the possible data items (instead of –inf to +inf)
  - Has same arithmetic operations available
- What's the relationship/difference between the ADT and the Implemented Data Type in Python/Java/C++?
  - The range of possible data items is different.

To summarize

- Abstract Data Type
  - E.g., the mathematical class I in our example
- Actual Implemented Data Type
  - What you have in Python, for example, “int” in our example
- Instantiated Data Type/Data Structure
  - E.g., x in int x = 5;
  - Stores (or structures) the data item(s)
  - Can be a variable, array, object, etc.; holds the actual data (e.g., a specific value)

Implementation of an ADT

- The data structures used in implementations are EITHER already provided in a programming language (primitive or built-in) or are built from the language constructs (user-defined).
- In either case, successful software design uses data abstraction for
- **SEPARATING THE DECLARATION OF A DATA TYPE FROM ITS IMPLEMENTATION.**

ADT Basics

- **All about designing a data type that others can use easily!**
- Two parts
  - A set of data (struct)
  - Operations on that data (methods)
- Separation between use and implementation
  - We're concerned with the implementation of ADTs
- For Java/C++ programmers, very similar to classes
  - (Optional) Look for Object Oriented design principles (notes in the class web site)

Why use ADTs?

- The separation of use and implementation increases portability of code
  - Example: Complex Number ADT
    - One person will write the data type, but ANY other programmers can use the data type in their code!
    - Simple as a declaration of any other primitive (int, float, char, double) data type

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:

`make_rat``(n, d)` – *returns the rational number* $\frac{n}{d}$

`numer``(x)` – *returns the numerator of* x

`denom``(x)` – *returns the denominator of* x

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)
```

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
```

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.

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

Using the ADT

Abstraction Barrier

make_rat, numer, denom

Implementing the ADT

RATIONAL NUMBERS AS TUPLES

Using the ADT

Abstraction Barrier

tuple, getitem

Implementing the ADT

Data Abstraction: So What?

It makes code more readable and intuitive.

```
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]
```

#IT WILL STILL WORK!

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

#IT WON’T WORK ANYMORE!

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

Practice: Data Abstraction

Read the following function prod_rats that takes a tuple of rational numbers using our ADT and returns their product. Correct the code removing 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

```
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

Definition of a **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

```
def make_student(name, id, age):
    return (name, id, age)
def student_name(s):
    return s[0]
def student_id(s):
    return s[1]
def student_age(s):
    return s[2]
```

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.

Object Oriented Programming in general

- Break your program into types of Objects
  - Player
  - Enemy
  - Map
  - Application
  - ...
- Collect all data and functions for that type into a class.
  - Example (Player):
    - Data: position, health, inventory, ...
    - Functions: draw, handleInput, update, hitDetect, ...
- Most modern software engineers use this approach to programming

Classes and Objects

- **Classes**
  - A blueprint for a new python type
  - Includes:
    - Variables (attributes)
    - Functions (methods)
- **Objects (instances)**
  - A variable built using a class "blueprint"
  - All python variables are objects.
  - Creating an object creates a new set of attributes for that object.
  - You call methods of the class through an instance.

Example

```
class Player(object):
    """This is the docstring for the player class."""
    def attack(self):
        """ This is the docstring for a method."""
        print("HIII-YA!“)

P = Player()    # This creates a new instance of Player
Q = Player()    # Another instance
P.attack()      # This calls the attack method of P.
Q.attack()      # self will refer to Q here.
```

Example

- **self parameter:**
  - when we call P's attack method, we don't pass an argument for self.
  - **self is a reference (alias) to the calling object**
    - While we're in the attack method called from line 9, self refers to P.
    - While we're in the attack method called from line 10, self refers to Q.
  - Both calls use the same method definition, but self refers to different things.
  - All "normal" methods will have self as their first parameter.

Adding attributes.

To really do something useful, the class needs data (attributes).
The first (bad) way of doing it would be:

```
class Player(object):
        """This is the docstring for the player class."""
        def attack(self):
                """ This is the docstring for a method."""
                print("HIII-YA!")
                print(self.name + " says 'HIII-YA!'")

P = Player()    # This creates a new instance of Player
Q = Player()    # Another instance

P.name = "Bob"  # Associates the attribute (variable) name with P
P.attack()      # This calls the attack method of P.
Q.attack()      # Error!
```

Problem: we have to "manually" add attributes to all instances (error-prone)

Adding attributes the "right" way (constructors)

- Python allows you to write a special method which it will automatically call (if defined).
  - Called the constructor.
  - In python it is __init__
- Example:

```
class Player(object):
    def __init__(self):
        print("Hi! I'm a new player")

P = Player()   # Calls __init__ (passing P) automatically
```

Adding attributes the "right" way (constructors)

- Pass arguments to __init__ and creating attributes.
- Example:

```
class Player(object):
      def __init__(self, newName):
            self.name = newName   # A new attribute
      def attack(self):
            print(self.name + " says 'HIII-YA!'")

#P = Player()        # Error.  __init__ requires an argument.
P = Player("Bob")   # Calls __init__ (passing P and      "bob")automatically.
Q = Player("Sue")
P.attack()
Q.attack()
```

Now we are guaranteed every player has a name attribute.

Temporary variables in methods

- Not all variables in a class method need self.
- Example:

```
class Player(object):
    def __init__(self, name):
        self.name = name
        self.hp = 100

    def defend(self):
        dmg = random.randint(1,10)
        self.hp -= dmg
```

string conversion of objects

- Normally, if you do this:
  - P = Player("Bob")
  - x = str(P)
  - print(x)
- you get output similar to:
  - <__main.Player object at 0x00A0BA90>
- This is the "default" way of converting an object to a string
  - Just like the "default" way of initializing an object is to do nothing.
- Python allows us to decide how to do it.

string conversion of objects

- Example

```
class Player(object):
        def __init__(self, newName):
            self.name = newName
        def __str__(self):
            return "Hi, I'm " + self.name

P = Player("Bob")
x = str(P)
print(x)   # Prints 'Hi, I'm Bob'
```

A common mis-conception: The __str__ method needs to return a string, not directly print it.

Questions?

@rschifan

schifane@di.unito.it

http://www.di.unito.it/~schifane