Part 4

CLASSES AS ABSTRACT DATA TYPES AND INTERFACE IMPLEMENTATION

ADT

ABSTRACT DATA TYPE

Class Definition as Abstract Data Type (ADT)

- **Abstract Data Type (ADT)**
  A programmer-defined data type that specifies a set of data values and a collection of **well-defined operations** that can be performed on those values
- Only the formal definition of the data type is important
- **NOT** how it is implemented in binary form or in hardware
- Sometimes called:
  "Separation of Interface and Implementation"

Information Hiding
Encapsulation

- **Hiding Information**
  How the data is represented?
  How the operations are implemented?
  This is completely **irrelevant** when we define a new **A**bstract **D**ata **T**ype (**ADT**) !!!
- **Encapsulation**
  A clear cut separation between developer view and client view. Class users should not know and should not care about how data is represented and how operations are implemented

Example: String ADT

String Data Type:

A string of characters like

```
s = "Hello World"
s = "Guido Van Rossum, 1993"
```

Operations:

upper(s)	All characters to upper case
lower(s)	All characters to lower case
find(s,w)	Find a word w in s (return index)
replace(s,w1,w2)	Replace sub word w1 with w2

EXAMPLE CODE:

```
s = "Hello World"
upper(s) = "HELLO WORLD"
lower(s) = "hello world"
find(s, "Wo") = 6
replace(s, "lo", " NEW") = "Hel NEW World"
```

ADT Implementations

- Note that the term ***"string of characters"*** does not imply anything about its implementation (how English characters are represented? How they are stored in memory? Disk?)
- It can be implemented as a **C: array of characters** terminated by a **NULL (each char in a byte, unicode,..)**
- It can be implemented like a Java or C++ String object
- We may even decide to encode and compress the string if it size is too large …
- We can decide to break each string to chunks of 4K in different memory locations and keep a central table for accessing these chunks (as a linked list), etc …
- In design decisions: The sky is the limit …

ADT As Interface Design

- Similarly, nothing on how the **find()** and **replace()** algorithms should be implemented is mentioned!
- Google's find() and replace() string algorithms are probably very different from those used in Microsoft Word !
- All we care is about how we **Interface** with the string data type? (How to do? instead of how it is done?)
- All implementation issues are **irrelevant** to the ADT specification!
- ADT specification is usually designed by clients
- ADT implementation designed by software developers

Container/Collection Terminology

- In Object Oriented Design, a **Container** is any object that contains other objects in itself
- Other words: a collection is a group of values with no implied organization or relationship between the individual values
- Some languages restrict the elements to a specific data type such as integers or floating-point values
  - Python collections do not have such restriction!
  - A Python collection may contain objects from mixed types!

Collection/Container Types

- The programming languages and literature are full with many such object with many different names
  - **List**
  - **Array**
  - **Sequence**
  - **Vector**
  - **Set**
  - **Stack**
  - **Queue**
  - **Heap**
  - **Map**
  - **Hash Table**
  - **Dictionary**
  - **Tree**
  - **Graph**
  - **Multimap**
  - **Multiset**
  - **Priority Queue**
  - **String**

Leaf Objects

- In contrast to Container object, a Leaf Object is an object that does not contain any reference to other objects (“has no child objects”)
- In Python these are sometimes called “primitive types”
  - Integer
  - Float
  - Complex number
  - Boolean
  - Char
- Leaf Objects are the building blocks from which all other objects are built

Primitive Types

- Integer: -5, 19, 0, 1000 (C long)
- Float: -5.0, 19.25, 0.0, 1000.0 (C double)
- Complex numbers: a+bj
- Boolean: True, False
- Long integers (unlimited precision)
- Immutable string: “xyz”, “Hello, World”

Arithmetic Operations

Operation	Result
x + y	sum of *x* and *y*
x - y	difference of *x* and *y*
x * y	product of *x* and *y*
x / y	quotient of *x* and *y* (Integer division if x,y integers
x % y	remainder of x / y
-x	*x* negated
+x	*x* unchanged
abs(x)	absolute value or magnitude of *x*
int(x)	*x* converted to integer
long(x)	*x* converted to long integer (this is very long …)
float(x)	*x* converted to floating point
complex(re,im)	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero
c.conjugate()	conjugate of the complex number *c*. (Identity on real numbers)
divmod(x, y)	the pair (x / y, x % y)
pow(x, y)	*x* to the power *y*
x ** y	*x* to the power *y*

Comparisons

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
=>	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Bitwise Operations

Operation	Result
x | y	bitwise *or* of *x* and *y*
x ^ y	bitwise *exclusive or* of *x* and *y*
x & y	bitwise *and* of *x* and *y*
x << n	*x* shifted left by *n* bits
x >> n	*x* shifted right by *n* bits
~x	the bits of *x* inverted

The Complex Numbers Class

- The **cmath** module defines Complex numbers arithmetic
- Python contains a built-in type (class) for complex numbers
- A complex number object has two fields and one method:

imag	imaginary part
real	real part
conjugate()	The conjugate number

```
import cmath
z = cmath.sqrt(-9)
⇒ 3j
z = cmath.sqrt(5-12j)
⇒ (3-2j)
z.imag
⇒ -2.0
z.real
⇒ 3.0
z.conjugate()
⇒ (3+2j)
```

Abstract Data Types Operations

Constructors	Methods for creating new objects
Accessors	Methods for accessing internal data fields without modifying the data!
Mutators	Methods for modifying object data fields
Iterators	Methods for processing data elements sequentially

Test Driven Development

- In this highly recommended methodology you write your tests before the implementation of your ADT !!!
- After implementation, your tests should run and **PASS** after each modification you make to your implementation ("nightly test regression")
- The following tests are your "insurance policy" that your implementation is correct. The more tests you write, the better you're insured!

```
# Testing our Student ADT
def test():
    s = Student("Dany Cohen", "03125", "07/08/1985")
    assert s.get_name() == "Dany Cohen"
    assert s.get_id() == "03125"
    assert s.get_birthday() == "07/08/1985"
    s.set_name("Daniel Cohen")
    assert s.get_name() == "Daniel Cohen"
    s.set_birthday("08/07/1986")
    assert s.get_birthday() == "08/07/1986"
    print "PASSED"
```

Testing your implementation

- Remember: tests must be written before you even think about an implementation!
- Make sure your tests cover the major features
- After writing an implementation you must run your tests: if they fail, then your implementation is bad
- After changing an implementation you must run all the tests again !!
- You may decide to throw away the whole implementation and write **a new one**, without any change to your ADT specification (“same Interface, different implementation”) – your tests should pass again with the new implementation!

ADT Implementation

- After defining an abstract data type, we need to implement it in a specific programming language
- First we must define a concrete data structure in the particular language for representing our abstract data
- Python basic data structures are usually implemented in the C programming language
- More complex data structures are usually implemented over the Python languages itself, and later transformed to C code if performance is critical

Interface and Implementation
Totally Separated Things !!!

- There should be a **total separation** between an **ADT** specification (sometimes called “Interface specification”) and its possibly many implementations
- For example, the Python Language has a full implementation over Java (called **Jython**), and at the same time Microsoft has a full implementation of Python over C# which is called **IronPython**
- The Python implementation over **C** is called **CPython**
- The same **Python tests** must all pass in all three implementations: **CPython**, **Jython**, and **IronPython** !
- The Python language itself is a pure interface! Unlike low level languages such as C it does not have any business with hardware registers, contiguous memory cells, etc. No relation to hardware at all!

Problems with Procedural Design

- No clear separation between major and minor data types
- For example, when we see `append(a,b)` it's not always clear which is the list and who is the element?
- Composite expressions like:

  ```
  insert(append(extend(L1,L2),a3),7,b4)
  ```

  can be very hard to read and understand compared to:

  ```
  L1.extend(L2).append(a3).insert(7,b4)
  ```

- Generic method names like `append()`, `insert()`, `remove()`, `size()`, etc., cannot be reused for a different data structure (like `FILE` or `Vector`), since they are global and already taken by the List data type … this is a serious trouble.
- Code reuse is difficult

The List ADT [As an Example]
Object Oriented Design – part 1

- `L = list_create1(e0, e1, e2,..., en-1)` [constructor]
  - Create a new list `L` from n elements: `e0, e1,... , en-1`
- `L = list_create2(other)` [constructor]
  - Create a new list **L** from other list or a container structure [conversion]
- `L.item(i)` - Get element `i` of list `L` [accessor]
- `L.contains(e)` [accessor]
  - Check if element **e** belongs to list `L`
  - Returns: boolean `True` or `False`
- `L.append(e)` [mutator]
  - Add a new element `e` to `L`
  - What if **e** already belongs to `L`? (answer: duplications are allowed!)
- `L.remove(e)` [mutator]
  - Remove an element `e` from `L`
  - What if `e` is not in `L`? (two possibilities: 1. do nothing, 2. raise an error)

The List ADT
Object Oriented Design – part 2

- `L.replace(index, e)` [mutator]
  - Replace element at index **`index`** with **`e`**
- `L.insert(index, e)` [mutator]
  - Insert a new element **`e`** at index **`index`**
  - Side effect: list grows by one element
- `L.size()` [accessor]
  - Return the size of **`L`**
- `L.extend(L2)` [mutator]
  - Extend list **`L`** by list **`L2`**
- `L.reverse()` [mutator]
- `L.slice(i,j)` [accessor]
  - Return a sub-list consisting of all elements of **`L`** from index **`i`** to index **`j-1`**
- `L.index(e)` [accessor]
  - Find the index of element **`e`** in **`L`**

Test Driven Development

- Before implementing the List ADT, we write a test!
- Our implementation must pass this test to be qualified as correct!

```
# Testing our List ADT
L1 = list_create1(2,3,5,7,11)
L2 = list_create2(L1)     # “copy constructor”
assert L2 == L1           # Assertion
assert L2.item(0) == 2
L1.append(37)
L1.remove(2)
L1.remove(3)
L3 = list_create1(5,7,11,37)
assert L1 == L3           # Assertion
assert L3.index(37) == 3  # Assertion
L3.reverse()
L4 = list_create1(37,11,7,5)
assert L3 == L4           # Assertion
```

If it isn’t tested it doesn’t work !!

Procedural notation

- The functional notation
  `foo(x)`, `bar(x,y)`, `baz(x,y,z)`
  was invented by the Mathematician Leonard Euler at 1748
- There is no specific sacred or holly reason for this notation! Euler could at the same time use ‘<x>f’ or ‘f-x-’ or many other possible notations
- We already have exceptions to this rule when we write x+y instead of add(x,y), or x**n instead of power(x,n).
- Python writes: `L = [a, b, c]` instead of `list_create(a,b,c)`

Python List Constructors

- The most basic constructor for lists is:

```
L = [x0, x1, x2, ..., xn]
```

  - It corresponds to: `list_create1(x0, x1, x2, …, xn)`
- The other constructor is **list(container_object)**
- Lists can be created from a variety of other container objects such as: set, array, string, dictionaries, and other lists

Naming Issues

- Specification name and Implementation name do not have to be the same!
- For example, in Python, the call

  ```
  L = list_create1(e0, e1, e2,..., en-1)
  ```

  has been changed to:

  ```
  L = [e0, e1, e2, …, en-1]
  ```

  and the call

  ```
  L.contains(e)
  ```

  Has been changed to:

  ```
  e in L
  ```

- The only essential thing is that the **name** conveys the **meaning** of the operation, and the operation is precisely defined

Python List Syntactic Sugar

Operation	Python Syntactic Sugar
`L=list_create1(a,…,b)`	`L = [a, ...,b]`
`L=list_create2(other)`	`L = list(other)`
`L.contains(e)`	`e in L`
`L.item(i)`	`L[i]`
`L.size()`	`len(L)`
`L.slice(i,j)`	`L[i:j]`
`L.equals(other)`	`L == other`
`L.remove_by_index(i)`	`del L[i]`
`L1.add(L2)`	`L1+L2`
`L.mul(n)`	`L*n  or  n*L`

A Word About Destructors

- Some object oriented languages (like C++) contain an additional method type: **destructor**
- A **destructor** is a method for destroying (or terminating) an object
- A destructor usually frees the memory that was used by the object and may also perform additional cleanup and finalization tasks
- In such languages, failure to delete objects at the right time can lead to serious memory problems, and even to program crash
- Modern object oriented languages such as **Java**, **C#**, and **Python**, contain a mechanism (called “garbage collection”) which automatically deletes objects as soon as they’re not needed anymore
- We will therefore not bother about this concept anymore in this course
- In extreme cases if needed you can use the Python del operator to delete objects: `del L`