Part 2

DICTIONARY, SET, TABLE

Set Abstract Data Type

- A set data structure is a container of objects with the following properties
- Elements are unique. A set cannot contain two instances of the same element (like a list or an array)
- Elements do not have an order. All we know about an element **e** is whether it belongs or does not belong to a set
- Set data structure originate in the mathematical theory of Set Theory, but have useful applications in computer science

The Set Abstract Data Type
Object Oriented Design – Part 1

- `s = set_create1()`
  - Create a new empty set s
- `s = set_create2(container)`
  - Create a new set s from other set or any another container object
- `s.add(e)`
  - Add element e to set s
- `s.remove(e)`
  - Remove an element `e` from the set `s`
  - If e is not in s, raise an error
- `s.contains(e)`
  - Check if element `e` belongs to the set `s`
  - Returns: boolean `True` or `False`
  - Efficiency requirement: should be very fast! O(1)

The Set Abstract Data Type
Object Oriented Design – Part 2

- `s.union(container)`
  - Set union of s elements with elements in container
  - Container can be any Python container (including a dictionary!)
  - Does not modify s! Just return the result!
- `s.intersect(container)`
  - Intersection of s with any other Python container
  - Does not modify s! Just return the result!
- `s.subtract(container)`
  - Remove from s all elements in container
- `s.discard(e)`
  - Remove an element from a set if it is a member
  - If the element is not a member, do nothing
- `s.clear()`
  - Remove all elements of s (make s an empty set)

The Set Abstract Data Type
Object Oriented Design – Part 3

- `s.copy()`
  - Create a copy of s
  - Same as: `s2 = set(s)`
- `s.issubset(container)`
  - Check if s is a subset of container. Return: `True` or `False`.
  - Container can be any Python container (even a dictionary!)
- `s.isdisjoint(container)`
  - Check if `s` is disjoint to container (no common elements)
- `s.issuperset(container)`
  - Check if s includes container elements. Return: `True` or `False`.
- `s.pop()`
  - Remove an arbitrary element from `s`
  - Raise an error if `s` is empty

The Set Abstract Data Type
Object Oriented Design – Part 4

- `s.equal(s2)`
  - check if two sets are equal (same as: s == s2)
- `s.update(container1, container2, ..., containern-1)`
  - Add elements from other containers
- `s.iterator()`
  - Create an iterator object for iterating over the set elements
- `s.size()`
  - Get the size of s (number of elements)

Set Test

```
s1 = set_create1()
s1.add(17)
s1.add(18)
s1.add(18)                  # adding 18 twice!
assert s1.contains(17)
assert s1.size() == 2
A = list_create1(2, 4, 6, 8, 2, 6)    # list container
B = list_create1(4, 8, 2, 6)          # list container
s2 = set_create2(A)
s3 = set_create2(B)
assert s2.equals(s3)
s3.add(100)
assert s2.issubset(s3)
s3.remove(100)
assert s2.equals(s3)
```

This is just a small example of how ADT regression test should look like.
A real test should cover all the ADT operations from all possible angles.
After every implementation change, the test should pass.

Set Implementation as List

- Python set is already implemented as a C hash table
- But it could also be implemented by the standard Python List data structure
- The implementation is available at this link:
  Link to Set implementation as list
- You also need to download
  Link to three set tests

The Dictionary (Map) ADT
Object Oriented Design – Part 1

- The dictionary data structure store key/value pairs
- Its critical advantage is the speed for getting a value from a key! We'll later explain what O(1) is and why this is the fastest time
- `d = dict_create1()`
  - Create a new empty dictionary
- `d = dict_create2(key1: value1, key2: value2, ...)`
  - Create a new dictionary from a list of key/value pairs
- `d = dict_create3(map_object)`
  - Create a new dictionary from other map_object
- `d = dict_create4(iterable)`
  - Create a new dictionary from an iterator which returns key/value pairs

The Dictionary (Map) ADT
Object Oriented Design – Part 2

- `d.contains(key)`
  - Check if dictionary d contains a key
- `d.add(key, value)`
  - Adds a new key/value pair to the dictionary if the key is not already there
  - If the key already there, then the old value is replaced with the new value
- `d.remove(key)`
  - Remove key (and its associated value) from the dictionary
- `d.get(key)`
  - Get the value associated with key
- `d.iterator()`
  - Creates and returns an iterator that can be used to iterate over the keys
- `d.copy()`
  - Copy a dictionary

The Dictionary (Map) ADT
Object Oriented Design – Part 3

- `d.clear()`
  - Remove all keys and values
- `d.items()`
  - Return a list of all key/value pairs stored in the dictionary
- `d.pop(key)`
  - Return the value associated with key, and remove key (and its associated value) from the dictionary
- `d.popitem()`
  - Remove an arbitrary key/value pair from the dictionary and return it
  - Raise an error if dictionary empty
- `d.update(map_object)`
  - Extend dictionary with additional key/value pairs from map_object

Dictionary Implementation – Part 1

- Python dictionaries are implemented as **hash tables**
- Hash tables are an advanced feature that we will cover at a later stage of the course, but here is a short summary of the idea which we learned from Brandon Rhodes Presentation on PyCon 2010
- A global Python hash function assigns to each key a 32 bit binary number which is called its hash value

```
key1 = 'Monty'
key2 = 'Money'
key3 = ('Hello', 'World', 2014)
key4 = 'Simplicity is the ultimate sophistication'
key5 = [1987, True, 'Hello']

hash(key1) = 01100111100110010110110011111110
hash(key2) = 00001101111101010011101110000101
hash(key3) = 11100100101100100001111011001111
hash(key4) = 01001100110010101100001110101010
hash(key5)
TypeError: unhashable type: 'list'
```

```
d = dict()
d[key1] = ‘foo’
d[key2] = ‘bar’
d[key3] = (1987,12)
D[key4] = [1,2,3]
```

Dictionary Implementation – Part 2

- A new dictionary is always created with 8 entries
- A dictionary is organized in memory exactly as a contiguous array of `(index, hash, key, value)` structures
- **Key retrieval:** get last 3 bits of hash(key) (`hash(key)&7`) and look up for the corresponding entry in the table array

index	hash	key	value
000			
001			
010	01001100110010101100001110101010	'Simplicity is the ultimate sophi…'	[1,2,3]
011			
100			
101	00001101111101010011101110000101	‘Money’	‘bar’
110	01100111100110010110110011111110	‘Monty’	‘foo’
111	11100100101100100001111011001111	('Hello', 'World', 2014)	(1987,12)

Dictionary Implementation – Part 3

- **Key insertion:** get last 3 bits of hash(key). If the corresponding row in the table is vacant, just insert the index, hash, key, and value a
- Key collision: if the row is already occupied with an item with the same index, we'll place it on the next vacant row
- For example, if we define anew entry:

  `d['ort'] = 'braude'`

  we find that

  `hash('ort') = '1111001000011001010010101100101010'`

  But we already have a row for **010** in our table, so we look for a new row by the recurrent formula: j = (5*j + 1) mod 8. In our case j=2, the new row is j=3

index	hash	key	value
000			
001			
010	01001100110010101100001110101010	'Simplicity is the ultimate sophi…'	[1,2,3]
011	11110010000110010100101011001010	'ort'	'braude'
100			
101	00001101111101010011101110000101	'Money'	'bar'
110	01100111100110010110110011111110	'Monty'	'foo'
111	11100100101100100001111011001111	('Hello', 'World', 2014)	(1987,12)

Dictionary Implementation – Part 4

- When the table is 2/3 full, we move into a 4-digits index and the table size is doubled!
- The 2/3 rule ensures we don't spend too much time on key lookup
- The full C implementation of Python dictionary can be viewed at:
  http://svn.python.org/projects/python/trunk/Objects/dictobject.c

index	hash	key	value
0000			
0001	10110101111111110100101010000001	'club 54'	'New York'
0010			
0011			
0100			
0101	00001101111101010011101110000101	'Money'	'bar'
0110			
0111			
1000			
1001			
1010	01001100110010101100001110101010	'Simplicity is the ultimate sophistication'	[1,2,3]
1011	11110010000110010100101011001010	'ort'	'braude'
1100			
1101			
1110	01100111100110010110110011111110	'Monty'	'foo'
1111	11100100101100100001111011001111	('Hello', 'World', 2014)	(1987,12)

Dictionary Implementation – Part 5

- A **hash table** is an array of `PyDictEntry` structs
- The struct consists of 3 fields:
  - `me_hash` - this is the key hash index (me = map entry)
  - `*me_key` - a pointer to the key
  - `*me_value` - a pointer to the value
- This array doubles each time it grows to its full size
- Removing a key does not delete its struct, but leaves a dummy struct

```
typedef struct {
    Py_ssize_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;
```

Python Dictionary

- Python provides a very efficient and easy to use dictionary class
- There are two ways to create and initialize a Python dictionary
- Python dictionary has all the standard dictionary methods and more

```
# Create a new empty dictionary
d = dict()

# Create and initialize a dictionary
d = dict(name='Avi Cohen', age=32, id=5802231, address='Hayarden 43, Gedera')

# Alternative constructors:
# Create a new empty dictionary
d = {}

# Create and initialize a dictionary
d = {name: 'Avi Cohen', age: 32, id: 5802231, address: 'Hayarden 43, Gedera'}
```

Python Dictionary Methods

```
print("Avi's age is:", d['age'])
print("Avi's address is:", d['address'])
print("Avi has moved to a new town:“)
d['address'] = 'Hayarkon 25, Haifa'
del d[key]   # deletes the mapping with that key from d
len(d)       # return the number of keys
x in d       # return True if x is a key of d
x not in d   # return False if x is not a key of d
d.keys()     # returns a list of all the keys in the dictionary
d.values()   # returns a list of all the values in the dictionary
```

MultiSet Abstract Data Type

- A multiset is a set in which elements may occur several times
- Example: words in a text file. It's not enough to know the set of words, we're also interested in how many times each word occurs?
- As with set, multiset elements are not ordered. All we know about an element e is the number of times it appears
- In some implementations, the number of occurrences can be 0 and even negative !

The MultiSet Abstract Data Type
Object Oriented Design – Part 1

- `m = multiset_create1()`
  - Create a new empty set s
- `m = multiset_create2(container)`
  - Create a new set s from other set or any another container object
- `m.add(e, n=1)`
  - Add element e with n occurrences
- `m.remove(e)`
  - Remove an element `e` from the multiset `m`
  - Be silent If e is not in s (usual behavior)
- `m.contains(e)`
  - Check if element `e` belongs to the multiset `m`
  - Returns: boolean `True` or `False`
  - Efficiency requirement: should be very fast! O(1)

The MultiSet Abstract Data Type
Object Oriented Design – Part 2

- `m.subtract(container)`
  - Remove from s all elements in container
- `s.discard(e)`
  - Remove an element from a set if it is a member
  - If the element is not a member, do nothing
- `s.clear()`
  - Remove all elements of s (make s an empty set)

The Table Abstract Data Type

- The Table data type is the most important data type in the field of databases (“relational databases”), spread sheet software (like Microsoft Excel), and also in mathematics (for representing a **matrix** or a **two-dimensional array** of numerical data). In VLSI used for **Gate Arrays** and **FPGA**
- Data in a table is organized into **rows** and **columns**.
- Data element is accessed by two indices:
  - row index
  - column index
- This pair of indices (i,j) is called a cell

	A	B	C	D	E
1	1	15	11	5	12
2	4	4	29	2	2
3	6	54	4	7	88
4	21	1	8	40	11
5	14	5	19	2	4
6	2	11	10	6	17
7	7	24	8	7	60

The Table Abstract Data Type

- `t = Table(nrows, ncols)`
  - Create a new table with number of rows = **`nrows`**, number of columns = **`ncols`**
- `t.numRows()`
  - Return the number of rows in the table **`t`**
- `t.numCols()`
  - Return the number of columns in the table **`t`**
- `t.clear(value)`
  - Clear and set all elements to **`value`**
- `t.setitem(i, j, value)`
  - Sets (or modifies) the content of the cell **`(i,j)`**
  - Both indices must be within valid bounds: **`0<=i<nrows, 0<=j<ncols`**
- `t.getitem(i, j)`
  - Get the content of the cell **`(i,j)`**
  - Both indices must be within valid bounds: **`0<=i<nrows, 0<=j<ncols`**

“Syntactic Sugar”

- Most programming languages provide the more traditional syntax (based on a very long mathematical history)

```
v = t[i,j]   ⟺   v = t.getitem(i,j)
t[i,j] = v   ⟺   t.setitem(i,j,v)
```

sometimes round parens are used instead of brackets

```
v = t(i,j)   ⟺   v = t.getitem(i,j)
t(i,j) = v   ⟺   t.setitem(i,j,v)
```

- Providing shorter more intuitive syntax is sometimes called “Syntactic Sugar”

Table Indexing

	Column 0	Column 1	Column 2	Column 3
Row 0	T[0,0]	T[0,1]	T[0,2]	T[0,3]
Row 1	T[1,0]	T[1,1]	T[1,2]	T[1,3]
Row 2	T[2,0]	T[2,1]	T[2,2]	T[2,3]

3x4 table

“Syntactic Sugar” in C

- The C programming language supports multi-dimensional arrays (same type) but is using a different kind of syntactic sugar:

```
v = a[i][j]   ⟺   v = a.getitem(i,j)
a[i][j] = v   ⟺   a.setitem(i,j,v)
```

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Table Implementations - C

- The Table ADT can be implemented in several ways
- In C, a two dimensional array is implemented as an “array of arrays”

```
typedef struct {
    double value;
} cell ;

# Static allocation
cell table[30][40];

# Dynamic allocation
cell **table = (cell **)malloc(30 * sizeof(cell*)) ;
for (col = 0; col < 40; ++col)
    table[col] = (cell *)malloc(40 * sizeof(cell)) ;
```

Table Implementations - C

- As you can see, the C **two-dimensional array** requires a single type for all cells
- Table code must be duplicated for every new cell type
- The worst part is that it does not include Table methods
- Methods must be defined separately for every new cell type

```
# Implementing the 'clear' method:

void clear(cell **table, int numrows, int numcols, cell value)
{
    int row, col ;

    for(row = 0; row < numrows; row++)
        for(col = 0; col < numcols; col++)
            table[row][col] = value ;
}
```

Table Implementation 2 – C
Procedural Design

- Idea: cell (row,col) can be encoded by a single integer:

  `row * numcols + col`

- We therefore can represent a `numrows*numcols` table by a single **1-dimensional** array:

```
typedef struct {
    double value;
} cell ;

# Dynamic allocation of a 3x4 table
cell* table = (cell *)malloc(3*4 * sizeof(cell));
```

- In spite of the extra multiplication/addition needed for indexing, this approach has big advantage from a CPU cache point of view!

Table Python Implementation 1 (List)
Procedural Design

- Table can be implemented as a list of lists
- Cell values can be of any mixed types
- The numRows() and numCols() methods are easily defined as: len(table) and len(table[0])

```
table = [ [0,1,2,3] , [4,5,6,7] , [8,9,10,11] ]

# setitem method:
table[2][0] = 1978

def clear(table, value):
    numrows = len(table)
    numcols = len(table[0])
    for row in range(numrows):
        for col in range(numcols):
            table[row][col] = value
```

Table Python Implementation 1 (List)
Procedural Design

- This is essentially the same as the C “array of arrays” idea

```
table = [ [0,1,2,3] , [4,5,6,7] , [8,9,10,11] ]

# setitem method:
table[2][0] = 1978

def clear(table, value):
    numrows = len(table)
    numcols = len(table[0])
    for row in range(numrows):
        for col in range(numcols):
            table[row][col] = value
```

Table Python Implementation 2 (Dict)
Procedural Design

- Table can also be implemented as a dictionary whose keys are cell indices (row,col)
- We can use the dictionary to store additional information like the number of rows and columns:

```
def new_table(nrows, ncols, value=0):
    table = dict()           # table is a dictionary !
    for row in range(nrows):
        for col in range(ncols):
            table[row,col] = value
    table['nrows'] = nrows  # save num rows in dict !
    table['ncols'] = ncols
    return table
```

Table Python Implementation 2 (Dict)
Procedural Design

```
# setitem method:
# table[2][0] = 1978

def setitem(table, row, col, value):
    table[row,col] = value

def getitem(table, row, col):
    return table[row,col]

def numRows(table):
    return table['nrows']

def numCols(table):
    return table['ncols']
```

Table Python Implementation 2 (Dict)
Procedural Design

```
def clear(table, value=0):
    nrows = numRows(table)
    ncols = numCols(table)
    for row in range(nrows):
        for col in range(ncols):
            table[row,col] = value

def printTable(table):
    nrows = numRows(table)
    ncols = numCols(table)
    for row in range(nrows):
        for col in range(ncols):
            print "table[%d,%d] = %s" % (row, col, table[row,col])
```

Table Python Implementation 2 (Dict)
Procedural Design

- Download the table2.py file and run the test below

```
def test1():
    table = new_table(3,4)
    clear(table,17)
    table[0,0] = 40
    table[2,3] = 50
    printTable(table)
```

Table Python Implementation 3 (List)
Object Oriented Design

- To fully match the Table ADT we need to do it in an OOD way
- We will show two different ways:
  - List of lists representation
  - Dictionary representation
- There are of course many other ways to implement a Table ADT, some are more efficient, but the point of this discussion is to make a clear distinction between **Interface** and **Implementation**!

Class: Table
numRows()
numCols()
setitem(row,col,value)
getitem(row,col)
clear(value)

Table Python Implementation 3 (List)
Object Oriented Design

```
class Table:
    def __init__(self, nrows, ncols, value=0):
        self.nrows = nrows
        self.ncols = ncols
        self.list = list()
        for r in range(self.nrows):
            row = ncols * [value]
            self.list.append(row)

    def setitem(self, row, col, value):
        self.list[row][col] = value

    def getitem(self, row, col):
        return self.list[row][col]

    def numRows(self):
        return self.nrows

    def numCols(self):
        return self.ncols
```

Table Python Implementation 3 (List)
Object Oriented Design

```
class Table:

    # . . . continued

    def clear(self, value=0):
        for row in range(self.nrows):
            for col in range(self.ncols):
                self.list[row][col] = value

    def __str__(self):    # print method !
        tbl = ""
        for row in range(self.nrows):
            for col in range(self.ncols):
                tbl += "table[%d][%d] = %s, " % (row, col, self.list[row][col])
            tbl += "\n"
        return tbl
```

Table Python Implementation 3 (List)
Object Oriented Design

- Here is a small test for testing our class
- Of course, a real life test should be more extensive !
- Download the source code and run the test

```
def test1():
    table = Table(4,5)
    table.clear(17)
    table.setitem(0,0,40)
    table.setitem(3,2,80)
    print table
    print "Number of rows =", table.numRows()
    print "Number of columns =", table.numRows()
```

Table Python Implementation 4 (Dict)
Object Oriented Design

```
class Table:
    def __init__(self, nrows, ncols, value=0):
        self.nrows = nrows
        self.ncols = ncols
        self.dict = dict()
        for row in range(self.nrows):
            for col in range(self.ncols):
                self.dict[row,col] = value

    def setitem(self, row, col, value):
        self.dict[row,col] = value

    def getitem(self, row, col):
        return self.dict[row,col]

    def numRows(self):
        return self.nrows

    def numCols(self):
        return self.ncols
```

Table Python Implementation 3 (List)
Object Oriented Design

```
class Table:

    # . . . continued

    def clear(self, value=0):
        for row in range(self.nrows):
            for col in range(self.ncols):
                self.dict[row,col] = value

    def __str__(self):                  # print method !
        tbl = ""
        for row in range(self.nrows):
            for col in range(self.ncols):
                tbl += "table[%d,%d] = %s, " % (row, col, self.dict[row,col])
            tbl += "\n"
        return tbl

    def __setitem__(self, key, value):   # overload the [] operator
        self.dict[key] = value

    def __getitem__(self, key):          # overload the [] operator
        return self.dict[key]
```

Table Python Implementation 3 (List)
Object Oriented Design

- Same test1() from implementation 3 should give identical result!
- We also add a test2() for testing the brackets overloading

```
def test1():
    table = Table(4,5)
    table.clear(17)
    table.setitem(0,0,40)
    table.setitem(3,2,80)
    print table
    print "Number of rows =", table.numRows()
    print "Number of columns =", table.numRows()

def test2():
    table = Table(4,5)
    table.clear(17)
    table[0,0] = 40
    table[3,2] = 80
    print table[3,2]
    print table
```