

SUPSI

Sets and Maps

Sets and maps are predefined in Python

- Sets are called just like that
 - `myset = set([1,2,3])`
- Maps are called dictionaries
 - `mymap = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}`
- Yet, it is important to analyse how they work and how they are defined as they might not be provided in other languages

Sets

Sets

- The set is a very common container
 - unlike a list it contains only unique values
 - As their mathematical counterpart they allow for useful operations:
 - intersection,
 - union,
 - subset,
 - difference.
- A set is a container that stores a collection of **unique values** over a given **comparable domain** in which the stored values have **no particular ordering**.

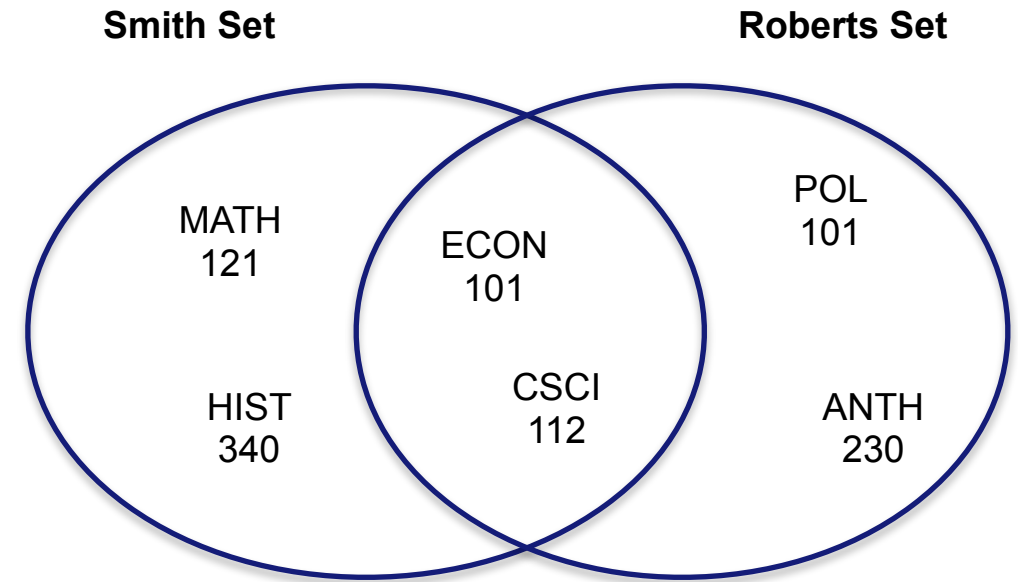
The Set ADT

- `Set()`: Creates a new set initialized to the empty set.
- `length()`: Returns the number of elements in the set, also known as the cardinality.
- `contains(element)`: Determines if the given value is an element of the set and returns the appropriate boolean value. Accessed using the `in` operator.
- `add(element)`: Modifies the set by adding the given value or element to the set if the element is not already a member. If the element is not unique, no action is taken and the operation is skipped.
- `remove(element)`: Removes the given value from the set if the value is contained in the set and raises an exception otherwise.
- `equals(setB)`: Determines if the set is equal to another set and returns a boolean value. Access with `==` or `!=`.
- `isSubsetOf(setB)`: Determines if the set is a subset of another set and re-turns a boolean value.
- `union(setB)`: Creates and returns a new set that is the union of this set and setB. Neither set A nor set B is modified by this operation.
- `intersect(setB)`: Creates and returns a new set that is the intersection of this set and setB. Neither set A nor set B is modified by this operation.
- `difference(setB)`: Creates and returns a new set that is the difference of this set and setB. The set difference, $A-B$, contains only those elements that are in A but not in B. Neither set A nor set B is modified by this operation.
- `iterator()`: Creates and returns an iterator that can be used to iterate over the collection of items.

Example of use

```
smith = Set()
smith.add( "CSCI-112" )
smith.add( "MATH-121" )
smith.add( "HIST-340" )
smith.add( "ECON-101" )
roberts = Set()
roberts.add( "POL-101" )
roberts.add( "ANTH-230" )
roberts.add( "CSCI-112" )
roberts.add( "ECON-101" )
```

```
if smith == roberts :
    print( "Smith and Roberts are taking the same courses." )
else:
    sameCourses = smith.intersection( roberts )
    if sameCourses.isEmpty() :
        print( "Smith and Roberts are not taking any of "\
            + "the same courses." )
    else:
        print( "Smith and Roberts are taking some of the "\
            + "same courses:" )
    for course in sameCourses :
        print( course )
```



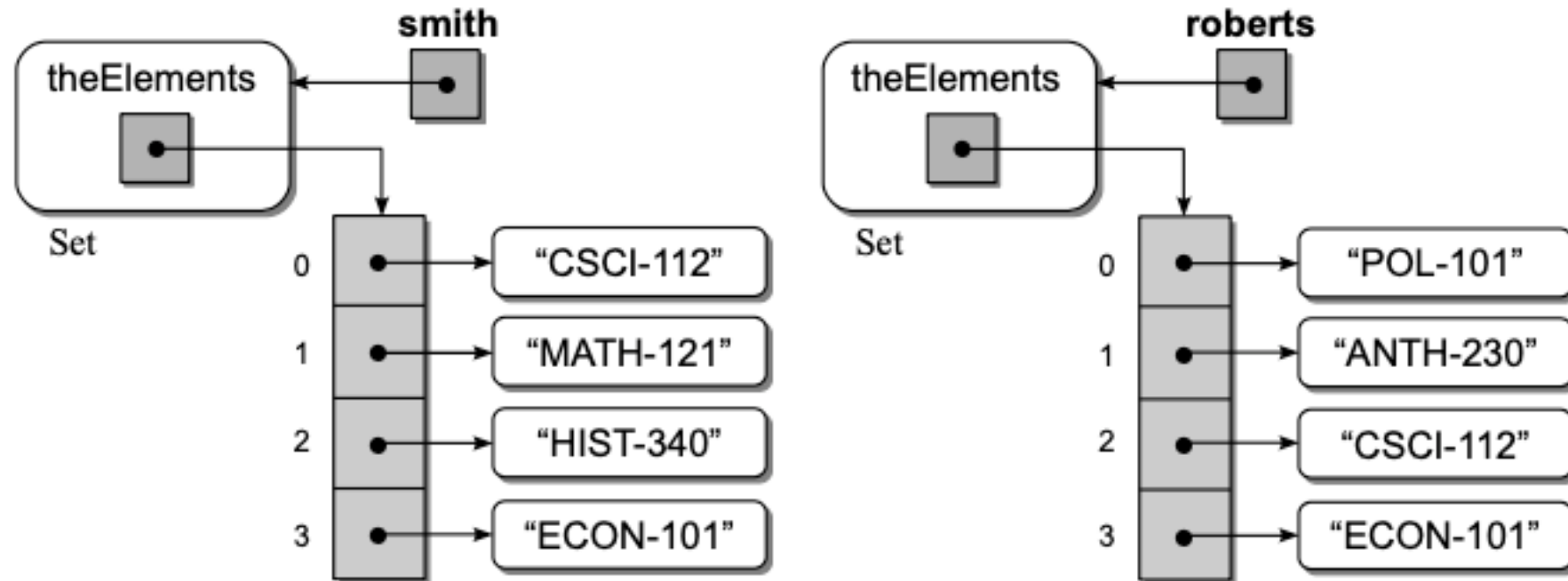
```
print( "The courses that only smith is "\
    taking are" )
uniqueCourses = smith.difference( roberts )
for course in uniqueCourses :
    print( course )
```

Selecting a data structure

- The most logical choice would be to use the `set()` pre-defined in Python, but the purpose here is to make a tough experiment on how would we implement the Set had it not been already provided by Python
- We can choose among:
 - Arrays
 - Lists
 - Dictionaries
- Dictionaries automatically store unique keys, but that would be a waste of space (need to store both key and value)
- Arrays do not expand automatically
- Lists are fine, but we must ensure that we won't store duplicates

List-based implementation

- Two Set() instances implemented as lists



Implementing the Set ADT (1)

```
class Set:
    # Creates an empty set instance.
    def __init__(self):
        self._theElements = list()

    # Returns the number of items in the set.
    def __len__(self):
        return len(self._theElements)

    # Determines if an element is in the set.
    def __contains__(self, element):
        return element in self._theElements

    # Determines if two sets are equal.
    def __eq__(self, setB):
        if len(self) != len(setB):
            return False
        else:
            return self.isSubsetOf(setB)
```

Implementing the Set ADT (2)

```
# Adds a new unique element to the set.
def add(self, element):
    if element not in self:
        self._theElements.append(element)

# Removes an element from the set.
def remove(self, element):
    assert element in self, "The element must be in the set."
    self._theElements.remove(element)

# Determines if this set is a subset of setB.
def isSubsetOf(self, setB):
    for element in self:
        if element not in setB:
            return False
    return True
```

Implementing the Set ADT (3)

```
# Creates a new set from the union of this set and setB.
def union(self, setB):
    newSet = Set()
    newSet._theElements.extend(self._theElements)
    for element in setB:
        if element not in self:
            newSet._theElements.append(element)
    return newSet

# Creates a new set from the
def intersect(self, setB):
    ...

# Creates a new set from the
def difference(self, setB):
    ...
```

Maps

Maps

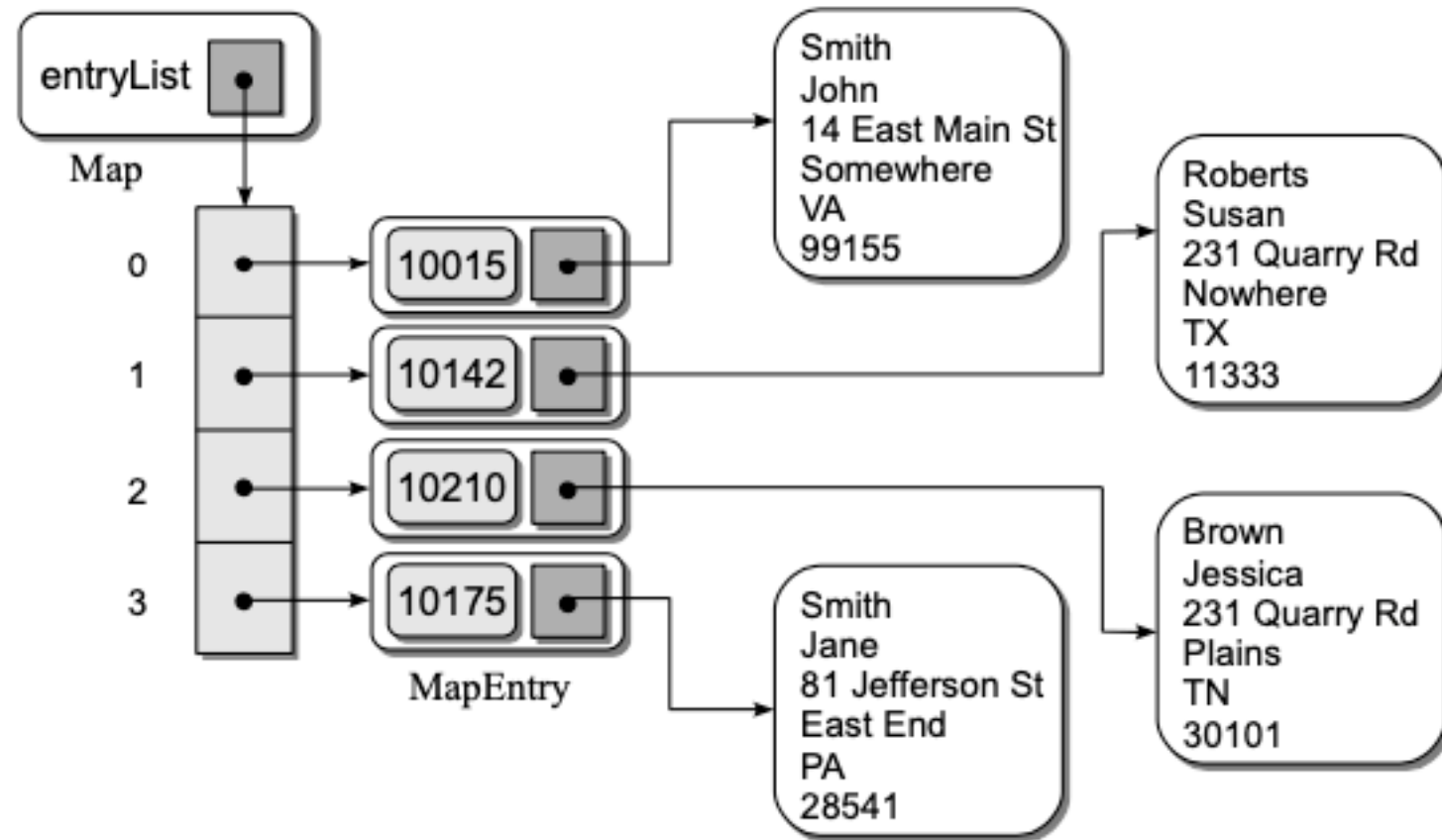
- A map allows to search for an item based on an index
- The data structure is usually called a **map** or a **dictionary**
- Typical problem: a list of individual records (e.g. student record) indexed by a unique identifier (the student's register id)
- Python provides the dictionary type which actually is a **hash map**
- Here we sacrifice performance for generality and we implement a Map that does not require a hash function
- Later in the course we will see how hash functions work

The Map ADT

- `Map()`: Creates a new empty map.
- `length()`: Returns the number of key/value pairs in the map.
- `contains(key)`: Determines if the given key is in the map and returns `True` if the key is found and `False` otherwise.
- `add(key, value)`: Adds a new key/value pair to the map if the key is not already in the map or replaces the data associated with the key if the key is in the map. Returns `True` if this is a new key and `False` if the data associated with the existing key is replaced.
- `remove(key)`: Removes the key/value pair for the given key if it is in the map and raises an exception otherwise.
- `valueOf(key)`: Returns the data record associated with the given key. The key must exist in the map or an exception is raised.
- `iterator()`: Creates and returns an iterator that can be used to iterate over the keys in the map.

Map ADT: a list based implementation

- We might need two lists, one for the key and one for the values
- The trick would be to maintain the association between the keys, in one list, and the values in the other parallel list
- Another approach relies on just one list, whose elements, the map entries, are key-values items



Map ADT implementation: the MapEntry

- The MapEntry allows for storing the key value pairs.

```
class _MapEntry:  
    def __init__(self, key, value):  
        self.key = key  
        self.value = value
```


Map ADT implementation: constructor and service

```
# Implementation of Map ADT using a single list.
class Map:
    # Creates an empty map instance.
    def __init__(self):
        self._entryList = list()

    # Returns the number of entries in the map.
    def len(self):
        return len(self._entryList)

    # Determines if the map contains the given key.
    def contains(self, key):
        ndx = self._findPosition(key)
        return ndx is not None
```

Map ADT implementation: add and remove

```
# Adds a new entry to the map if the key does exist. Otherwise, the  
# new value replaces the current value associated with the key.  
def add(self, key, value):  
    ndx = self._findPosition(key)  
    if ndx is not None: # if the key was found  
        self._entryList[ndx].value = value  
        return False  
    else: # otherwise add a new entry  
        entry = _MapEntry(key, value)  
        self._entryList.append(entry)  
        return True  
  
# Removes the entry associated with the key.  
def remove(self, key):  
    ndx = self._findPosition(key)  
    assert ndx is not None, "Invalid map key."  
    self._entryList.pop(ndx)
```

Map ADT implementation: valueOf and findPosition

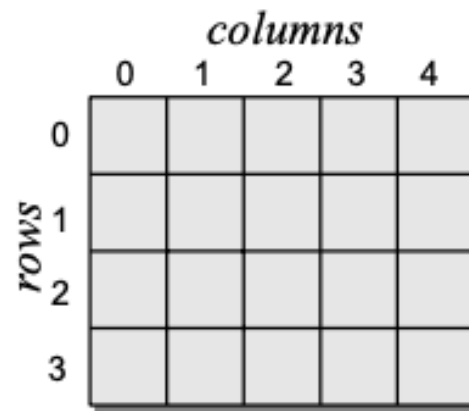
```
# Returns the value associated with the key.
def valueOf(self, key):
    ndx = self._findPosition(key)
    assert ndx is not None, "Invalid map key."
    return self._entryList[ndx].value

# Helper method used to find the index position of a category. If the
# key is not found, None is returned.
def _findPosition(self, key):
    # Iterate through each entry in the list.
    for i in range(len(self)):
        # Is the key stored in the ith entry?
        if self._entryList[i].key == key: return i
    # When not
    return None
```

Multi-dimensional arrays

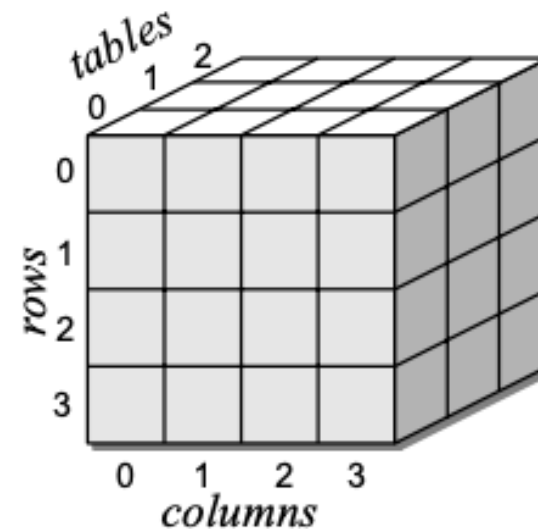
The multi-dimensional array

- A three-dimensional array can be visualised as a sequence of tables, where each table is a bi-dimensional array



A 2D array grid with 4 rows and 5 columns. The columns are labeled 0, 1, 2, 3, 4 at the top. The rows are labeled 0, 1, 2, 3 on the left. The label 'columns' is centered above the column headers, and 'rows' is centered to the left of the row headers.

	<i>columns</i>				
	0	1	2	3	4
0					
1					
2					
3					



A 3D array cube with 4 rows, 5 columns, and 3 tables. The columns are labeled 0, 1, 2, 3 at the bottom. The rows are labeled 0, 1, 2, 3 on the left. The tables are labeled 0, 1, 2 at the top. The label 'columns' is centered below the column headers, 'rows' is centered to the left of the row headers, and 'tables' is centered above the table headers.

						<i>tables</i>		
						0	1	2
0								
1								
2								
3								

- A four-dimensional array is harder to visualise, and it can be only imagined

Indexing n-dimensional arrays

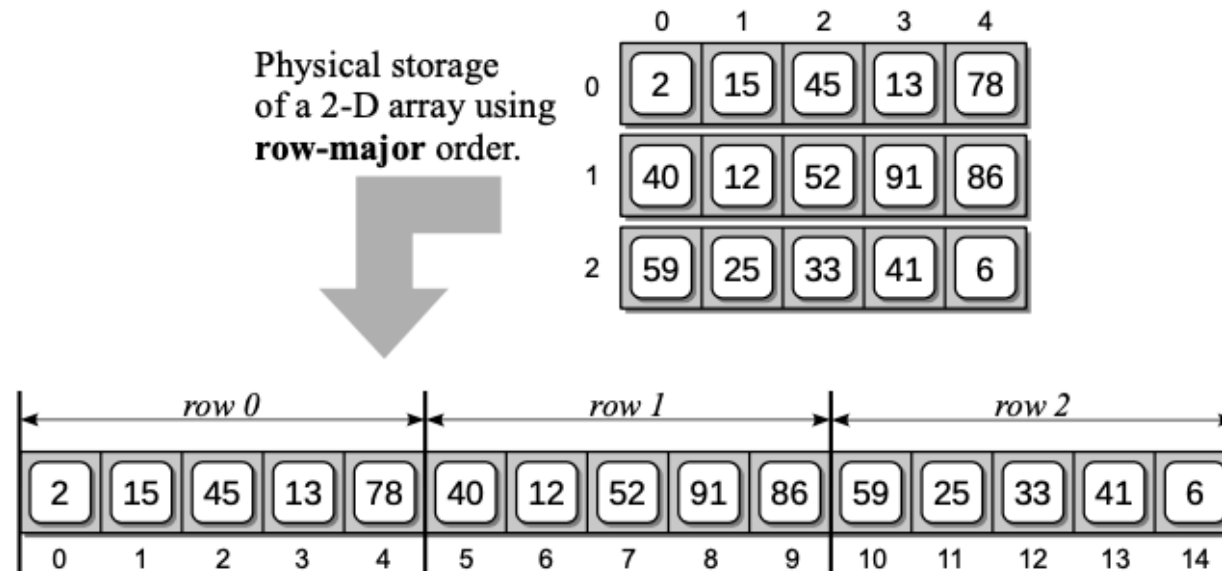
- Array x_i
 - Matrix $x_{i,j}$
 - Cube $x_{i,j,k}$
 - Hypercube $x_{i,j,k,m}$
 - ...
-
- Most languages provide programming structures to create and manage multi-dimensional arrays (e.g. C++ Java)
 - Python does not support arrays, but we can define the MultiArray ADT, just like we did the Array and Array2D

The MultiArray ADT

- `MultiArray(d1, d2, . . . dn)`: Creates a multi-dimensional array of elements organized into n-dimensions with each element initially set to `None`. The number of dimensions, which is specified by the number of arguments, must be greater than 1. The individual arguments, all of which must be greater than zero, indicate the lengths of the corresponding array dimensions. The dimensions are specified from highest to lowest, where `d1` is the highest possible dimension and `dn` is the lowest.
- `dims()`: Returns the number of dimensions in the multi-dimensional array.
- `length(dim)`: Returns the length of the given array dimension. The individual dimensions are numbered starting from 1, where 1 represents the first, or highest, dimension possible in the array. Thus, in an array with three dimensions, 1 indicates the number of tables in the box, 2 is the number of rows, and 3 is the number of columns.
- `clear(value)`: Clears the array by setting each element to the given value.
- `getitem i1, i2, . . . in)`: Returns the value stored in the array at the element position indicated by the n-tuple `(i1, i2, . . . in)`. All of the specified indices must be given and they must be within the valid range of the corresponding array dimensions. Accessed using the element operator: `y = x[1, 2]`.
- `setitem (i1, i2, . . . in, value)`: Modifies the contents of the specified array element to contain the given value. The element is specified by the n-tuple `(i1,i2,...in)`. All of the subscript components must be given and they must be within the valid range of the corresponding array dimensions. Accessed using the element operator: `x[1, 2] = y`.

Array storage

- In the Array ADT we have seen how we can use a C-based type to manage hardware storage of a linear array
- For multidimensional arrays the storage is software mapped to the hardware memory infrastructure which is linear.
- There are two main methods **row-major** and **column-major** order
- Most languages (exception is FORTRAN) use the row-major order



Index computation: 2D case

- If we follow the row-major order, whenever we want to access a given element we need to add an offset corresponding to the row where the element is stored:
 - If it is the first row, the offset is 0
 - In the second row the offset is the length of the row
 - In the third row it is 2 times the row length
 - In general, the offset is $i * \text{len}(\text{row})$ where i is the row index
- $\text{index}_2(i, j) = i * n + j$ is the location of element (i, j) of a 2-dimensional array in the corresponding 1-dimensional array

A 3x5 grid representing a 2D array. The columns are indexed from 0 to 4, and the rows are indexed from 0 to 2. The grid contains the following values:

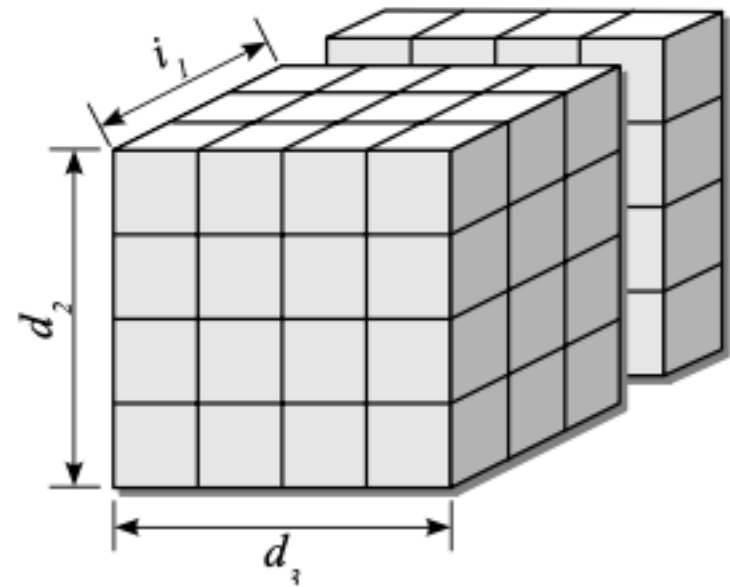
	0	1	2	3	4
0	2	15	45	13	78
1	40	12	52	91	86
2	59	25	33	41	6

The value 41 is highlighted in the cell at row index 2 and column index 3.

41 is element (2,3)

Indexing in higher dimensions

- Imagine we have a 3D array of dimensions $d_1 \cdot d_2 \cdot d_3$
- The 1-D array offset of element (i_1, i_2, i_3) using row-major order would be
 - $index_3(i_1, i_2, i_3) = i_1 \cdot (d_2 \cdot d_3) + i_2 \cdot d_3 + i_3$
- $d_2 \cdot d_3$ is the dimension of a table which must be skipped and the first index i_1 indicates how many such tables must be skipped
- The remainder of the formula is then like the access to an element in a 2D array: $i_2 \cdot d_3 + i_3$



Indexing on higher dimensions (2)

- We can therefore generalise:

$$\text{index}_4(i_1, i_2, i_3, i_4) = i_1 \cdot (d_2 \cdot d_3 \cdot d_4) + i_2 \cdot (d_3 \cdot d_4) + i_3 \cdot d_4 + i_4$$

- And we can extract a pattern!

$$\text{index}_n(i_1, i_2, \dots, i_n) = i_1 \cdot \prod_{k=2}^n d_k + i_2 \cdot \prod_{k=3}^n d_k + \dots + i_n \cdot 1$$

- The nice thing is that we can write this as

$$\text{index}_n(i_1, i_2, \dots, i_n) = i_1 \cdot f_1 + i_2 \cdot f_2 + \dots + i_n \cdot f_n \quad \text{with} \quad f_n = 1 \quad \text{and} \quad f_i = \prod_{k=i+1}^n d_k$$

Which is good since we can pre-calculate and store the f_i values (they don't change once the array dimension is set) which saves time and computation when accessing the array values

Variable length arguments

- The variable-length argument is needed for the constructor
 - `MultiArray(d1, d2, . . . dn)`
- And for the setters and getters
 - `setitem (i1, i2, . . . in, value)`
 - `getitem i1, i2, . . . in)`

```
def func(*args):  
    print ("Number of arguments: ", len( args ))  
    sum = 0  
    for value in args :  
        sum += value  
    print("Sum of the arguments: ", sum)
```

```
func( 12 )  
func( 5, 8, 2 )  
func( 18, -2, 50, 21, 6 )
```

The asterisk next to the argument name (*args) tells Python to accept any number of arguments and to combine them into a tuple.

Implementation of the MultiArray: constructor

variable length argument

```
# Implementation of the MultiArray ADT using a 1-D array.
class MultiArray:
    # Creates a multi-dimensional array.
    def __init__(self, *dimensions):
        assert len(dimensions) > 1, "The array must have 2 or more dimensions."
        # The variable argument tuple contains the dim sizes.
        self._dims = dimensions
        # Compute the total number of elements in the array.
        size = 1
        for d in dimensions:
            assert d > 0, "Dimensions must be > 0."
            size *= d
        # Create the 1-D array to store the elements.
        self._elements = Array(size)
        # Create a 1-D array to store the equation factors.
        self._factors = Array(len(dimensions))
        self._computeFactors()
```

Implementation of the MultiArray: computation of the factors

```
# Computes the factor values used in the index equation.
def _computeFactors(self):
    #f_n is 1
    self._factors[len(self._dims)-1]=1
    #f_i = \prod_{k=i+1}^{nd_k}
    for i in range(len(self._dims)):
        factor=1
        for j in range(i+1,len(self._dims)):
            factor=factor*self._dims[j]
        self._factors[i]= factor
    print("factor[" ,i,"]= ", factor)
```

Implementation of the MultiArray: getter and setter

```
# Returns the contents of element (i_1, i_2, ..., i_n).
def __getitem__(self, ndxTuple):
    assert len(ndxTuple) == self.numDims(), "Invalid # of array subscripts."
    index = self._computeIndex(ndxTuple)
    assert index is not None, "Array subscript out of range."
    return self._elements[index]

# Sets the contents of element (i_1, i_2, ..., i_n).
def __setitem__(self, ndxTuple, value):
    assert len(ndxTuple) == self.numDims(), "Invalid # of array subscripts."
    index = self._computeIndex(ndxTuple)
    assert index is not None, "Array subscript out of range."
    self._elements[index] = value
```

Implementation of the MultiArray: computation of the offset

```
# Computes the 1-D array offset for element (i_1, i_2, ... i_n)  
# using the equation i_1 * f_1 + i_2 * f_2 + ... + i_n * f_n  
def _computeIndex(self, idx):  
    offset = 0  
    for j in range(len(idx)):  
        # Make sure the index components are within the legal range.  
        if idx[j] < 0 | idx[j] >= self._dims[j]:  
            return None  
        else: # sum the product of i_j * f_j.  
            offset += idx[j] * self._factors[j]  
    return offset
```


Implementation of the MultiArray: utilities

```
# Returns the number of dimensions in the array.
def numDims(self):
    return len(self._dims)

# Returns the length of the given dimension.
def length(self, dim):
    assert dim >= 1 and dim <= len(self._dims), \
        "Dimension component out of range."
    return self._dims[dim-1]

# Clears the array by setting all elements to the given value
def clear(self, value):
    self._elements.clear(value)
```