What Data Structure to use ?

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Container Data Types

- These are data types whose objects are "containers" i.e. they have multiple objects within them
- These container types - have operations on the overall object, as well as allow access to the individual items
- Many container types are possible (e.g. stacks, queues, trees, ..) - and new container types can be defined in Python
- Python language has provided 4 built-in types: lists, sets, tuples, dictionaries - these are widely used
- An object of a type is created (in memory - heap) and the variable points to that object
- How objects of different types are implemented has impact on the efficiency of different operations on them

Objects in Memory

- Each object (int, float, string, list, set, …) is allocated some memory (on heap) by the runtime system
- Address to this memory is kept in the variable
- After (x = 55), when we access x, we get the address of the object x points to, and then get the value
- Compound/container objects - often a base size and then depends on the number of items
- Sometimes there is a memory-time tradeoff

Getting the size of an object

- We can get the size of an object by

  import sys

  sys.getsizeof(obj)
- Gives the size in bytes of obj
- The memory for obj is not just to store the value, but some other attributes (e.g. reference pointer etc)
- So, while an integer (till a limit) is stored in 4 bytes, an int object actually is assigned 28

Memory – Size of Objects

Lets check the size of some objects

int: 28

str: 49 +1 per additional character (49+total length of characters)

list: 56 (Empty List) +8 per additional item in a list (56 + 8*total length of characters)

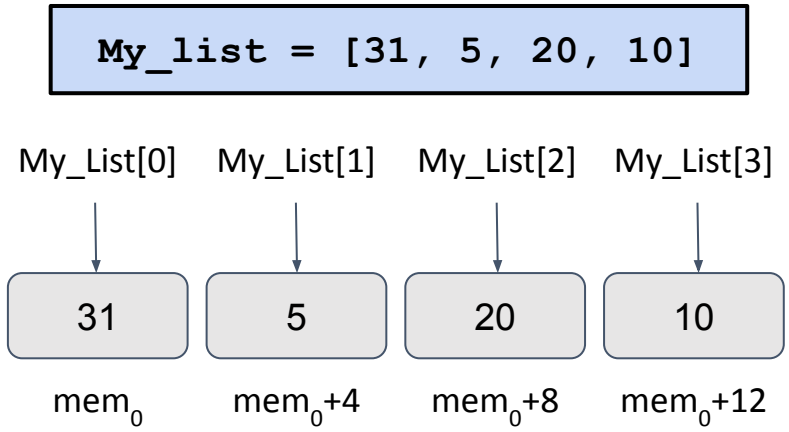
set: 216 (0-4 take the size of 216. 5-19 take size 728. 20th will take 2264 and so on...)

Execution Time

- Execution time depends largely on how many simple statements are executed from start to finish
- This depends on the logic/algorithm used to implement some operation; loops are the killer - execute statements many times
- Approaches / logic that need to execute too many statements take more time
- (Though some statements take more time than others, we will not worry about them)
- Sometimes there is a space-time tradeoff - though generally efficiency is about reducing execution time

List Implementation

- Lists are given contiguous memory - one for each item (the memory points to the item-object)
- Let us try to understand how efficient (fast/slow) are some of the operations on lists



List Operations

- Accessing i^{th} item
  - Address of the ith item is: addr of start-of-list + i * no-of-bytes-for-an-item-addr: this is a fast operation
  - Using the addr in the ith item, direct access the lst[i] object
  - Hence (access or update) ith element is ***fast*** operation
- Membership, i.e. `in` operation
  - To check if an item is a list, interpreter has to check items one by one
  - I.e. traverse the list till the item is found or till the end
  - This is ***slow*** operation
- Finding the index of an item
  - As objects can only be accessed through lst[i], it has to traverse list, then access object, and see if same
  - A ***slow*** operation

List Operations...

- insert(index, item)
  - Reaching index is fast
  - But has to add item at index, then shift all items after it by one
  - A ***slow*** operation
- append(item)
  - End of the list is kept/known
  - Adding at the end extend lst by one, and have it point to new object
  - No shifting - ***fast*** (extend also)
- pop(index) - have to shift items left - ***slow***,
- pop() is ***fast*** (as last item deleted, no shifting of items)

Set Implementation

- Sets have unique values (no duplicates) - so can use hashing
- In hashing, a contiguous memory, much larger than the number of items, is assigned
- An item value is converted by a hash function to an index in this large memory - index is used to access the item (which is fast)
- For using hashing on a collection of items, each item must be unique so its hashing returns a unique index
- This index is used to access the item, just like the ith item in a list - so is fast

Set implementation and indexing

- E.g. consider a set {12345, 45678, 90876, 6789}
- Suppose hash function is item%100 (i.e. the last two digits)
- A contiguous memory for 100 items will be assigned (each item will have a flag - exists/not exists, and value)
- Item 12345, will be in 45th location, 45678 in 78th ..
- So, accessing items, even though they are not ordered, due to indexing is direct access
- Hashing function is generally fast
- Fast hashing functions are used internally

Set Operations

- Membership checking for an item - hash the item to get index, check directly in the memory location - ***fast***
- Adding an item in set - check at index: hash(item) - if exists, then return, else, add the item in this - ***fast***
- Deleting an item - ***fast*** - just go to the index provided by hash
- So, set operations are fast - the cost is that it takes more memory (for a set of n items, the memory may be much larger than need for just keeping n items)

Set vs List

- Suppose you are to maintain a bag of items on which your main operations are to check its presence, add items, remove
- You can implement it as a list - it also provides in, add, delete/pop
- You can implement it as a set - in, add, delete
- Set will be much faster, but will take more memory than list

Quiz

Given a large number of unique integers. You want to sometimes add, delete another unique number to this group. You can maintain these values as a list or as a set. Which of these are true

A. . Adding to both list and set implementations will take similar time
B. . Deleting a specified number will be much faster in set than in list
C. . Both implementations will require similar amount of memory .
D. .Finding an element will be much faster in set than in list

Quiz – Answer

A, B, D

Dictionaries Implementation

- Dictionaries, like sets also use hashing on keys
- It also stores values along with keys
- The keys are hashed the same way as in Sets
- Thus, if we want to store 1324: “sam”, 4325: “tim”, 8945: “paul”
- With item%100 hashing function,
  - 1324 is hashed to 24
  - 4325 is hashed to 25
  - 8945 is hashed to 45

Index	**00**	**01**	**…**	**24**	**25**	**…**	**45**	**…**	**98**	**99**
Key				1324	4325		8945			
Value				sam	tim		paul			

Dictionaries Operations

- Dictionary keys are unique and so like sets:
  - ***fast*** implementation for access, insert, remove
  - But use more memory
- Note: accessing an item using a value instead of key will require traversing the dictionary - such operations not provided
  - You have to write code to traverse (key, value) pairs and find the index
- Dictionaries operations are generally ***fast*** (due to hashing)

Example: Removing Duplicates from a list

```
# Method 1
res1 = []
for i in tstl:
    if i not in res1:
        res1.append(i)

# Method 2
res2 = []
[res2.append(x) for x in tstl if x not in res2]

# Method 3
    res3 = list(set(tstl))
```

Which of these is most efficient?

Run these for lists of 10K size

Times (sec): 0.6, 0.4, 0.0007

Method 3 is extremely efficient

Tuples

- Tuples are like lists - ordered sequence of items (so indexes allowed), duplicates allowed (so hashing cannot be used), etc
- The main difference is that tuples are immutable
  - As number of tuple items is fixed, the interpreter can assign optimum memory - for lists it has to cater to expansion
  - A tuple of immutable items - these items cannot be changed, hence interpreter can assign them minimum memory needed
- Overall, tuples generally take a little less space than lists

Tuples

- As number of tuple items is fixed, the interpreter can assign optimum memory - for lists it has to cater to expansion
- A tuple of immutable items - these items cannot be changed, hence interpreter can assign them minimum memory needed
- Overall, tuples generally take less space than lists
- The method object.__sizeof__() gives the size of an object
  - Size of empty list is 40 (bytes)
  - Size of empty tuple is 24 (bytes)
  - Size of list/tuple of ints increase with items (8 bytes / item)
  - Size of list/tuple of same strings - different (due to some underlying optimization in storing items)

Note: Size may vary as per hardware

Quiz(Numerical)

What will be the output of the following code?

```
import sys
n = sys.getsizeof("abc")
m = sys.getsizeof("abcd")
print (m-n)
```

Quiz(Solution)

Solution: 1

Lists vs Tuple: Use

- Use tuples: when data is to be read/used only, and you don’t want the values to change (e.g. record of a student)
- Use lists: when the value of items need to change, when you need to move items around, add/delete items
- Use tuples where performance is critical (e.g. some very frequently used operation in a library)
- Lists are much more powerful - as more functions are provided
- For most applications, lists will work well

What Structure to Use?

- For many problems, the natural data structure will emerge
- If a collection of items with duplicates - lists and tuples
  - Use tuples: when data is to be read/used only, and you don't want the values to change (e.g. record of a student)
  - Use lists: when the value of items need to change, when you need to move items around, add/delete items, duplicates are permitted
  - Use tuples where performance is critical (e.g. some very frequently used operation in a library)
- Whenever unique values are to be collected - use sets
  - You can always convert them to lists (and back to sets) if some list operations are to be used
  - E.g. To remove duplicates from a list: convert a list to set to get rid of duplicates and convert back to list, is likely to be faster

Which DS to use ...

- Dictionaries are natural choice when we have a record with multiple fields, but with a unique key
- E.g student records to be stored - as unique roll no, which can be used as key - dictionary most natural (and fast)
- Naturalness - you will probably also find naturally needed operations provided to you
- You should also examine the efficiency aspects - which DS will provide a faster processing
- Generally, wherever you can use dictionary - use it as it is fast

Quiz(Single Correct)

You want to maintain a record of all the clients a company has worked with, in the order the clients were added. You need to be able to add and remove clients from the list. Which data structure should you use?

A. Lists

B. Dictionaries

C. Sets

D. Tuples

Quiz(Solution)

You want to store a list of all the clients a company has worked with, in the order the clients were added. You need to be able to add and remove clients from the list. Which data structure should you use?

A. Lists

B. Dictionaries

C. Sets

D. Tuples

Answer: A

Quiz – Single Correct

You have planned a seminar at your college and people have registered for it using their Email. Only registered participants are allowed to enter the seminar so whenever a participant arrives you need to check if their email address is present in your record, and then show all the data for that person. Which data structure would be most appropriate for maintaining this record?

A. List
B. Tuple
C. Set
D. Dictionary

Quiz – Single Correct

D. Dictionary

Impact of Algorithm on Performance

- The algorithm (logic) you use has huge impact on performance, particularly the time taken
- We looked at some algorithms for finding if a number N is a prime - you can divided till N-1, N/2, or sqrt(N) - different times
- Given a list of items - you need to often search if an item is present in the list
- You can keep the list - and search everytime (loop over the list)
- Or you can sort the list, and then search using binary search approach - check in the middle and then search the left or right
- Sorting can be expensive, but searching in a sorted list is very fast - so if searching is to be done often, you sort first

Frequency of items in a list

- Problem: Given a list of items, determine the frequency of different items in the list
- E.g. a list [1, 2, 3, 1, 1, 9, 2, 3, 5, 6, 7, 3, 1, 2, …]
- The output should give the frequency of occurrences of unique numbers: 1 occurs 4 times, 2 - 3 times, 3 - 3 times, 9 - 1 time, …
- Multiple ways to do it:
  - Keep two lists: unique items and their frequency - traverse the list and keep adding unique items and incrementing frequency count
  - Keep two lists: sort the input list first - then checking uniqueness is easy
  - Keep two lists: take each item - remove all its occurrences, keep repeating till the list is empty
  - Maintain a dictionary item: frequency
  - …

Determining frequency of items in a list

```
# Method 1 - use lists
def freq_order(lst):
    unique = []        # list of unique item
    freq =[]           # their frequencies
    for item in lst:
        if item in unique:      # already seen
            freq[unique.index(item)] += 1
        else:                   # new item
            unique.append(item)
            freq.append(1)
    return (unique, freq)
```

```
# Method 2 - use lists but sort input first
def freq_sort(lst):
    unique , freq = [], []
    lst.sort()
    unique.append(lst[0])
    freq.append(1)
    j = 0  # j represents index of unique items
    for i in range(1, len(lst)):
        if lst[i] == lst[i-1]:
            freq[j] += 1
        else:
            freq.append(1)
            unique.append(lst[i])
            j = j+1
    return(unique, freq)
```

```
# Method 3 - use dictionary
def freq_dict(lst):
    freq = {}        # empty dictionary
    for item in lst:
        if item in freq:       # already seen
            freq[item] += 1
        else:                  # new item
            freq[item] = 1
    return freq
```

- As frequency is of unique items - dictionary is a natural choice
- It is also the fastest - times are: 0.018, 0.007, 0.00005
- M2 is faster than M1, as sort is very efficient, and avoids the expensive membership test

Encryption

Caesar Cipher.

A very simple cipher where each letter is replaced by a letter which is fixed number of positions away from the current letter.

For example, shift of +3:

- A is replaced by D
- B is replaced by E
- C is replaced by F
- ...
- Y is replaced by B
- Z is replaced by C

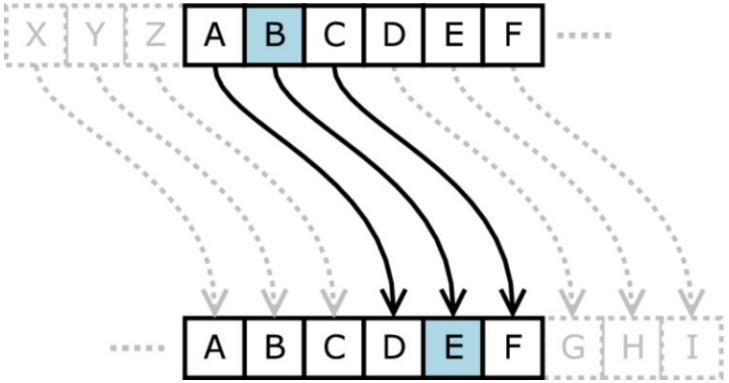


Image source

```
Text:    intro to programming
Encoded: lqwur wr surjudpplqj
```

Method 1 – Simple method

Idea: Traverse the input string and compute the replacement letter individually for each character.

```
def encrypt_simple(text, shift):
  # text: Input string. Assumption: Only lowercase letters, no space
  # shift: Integer between 0 to 25
  result = "" # String to store the final result after encryption

  # Traverse the string character by character and compute replacement
  for letter in text[:]:

    pos = ord(letter) - ord('a') # Get position of letter in alphabets
    new_pos = (pos+shift)%26 # Compute shift for letter
    replace = chr( new_pos + ord('a') ) # Replacement letter
    result += replace

  return result
```

Method 2 – Using list

Idea: Cut off the shifted part of the alphabet list from the front, and paste it to the back

```
def encrypt_list(text, shift):
  # text: Input string. Assumption: Only lowercase letters, no space
  # shift: Integer between 0 to 25

  alphabet = ['a','b','c','d', ..... , 'w','x','y','z']
  alphabet = [alphabet[(i+shift)%26] for i in range(len(alphabet))]
  # If shift=3, alphabet becomes = ['d','e','f','g', ..... , 'z','a','b','c']
  result = "" # String to store the final result after encryption

  for letter in text[:]:
     encrypted_letter = alphabet[ord(letter) - ord('a')]
     result += encrypted_letter

  return result
```

Method 3 –Using dictionary

Idea: Instead of computing shifts for each letter in input string repeatedly, we compute the shift for all 26 English alphabets once, and store them in a dictionary.

```
def encrypt_dict(text, shift):
    # text: Input string. Assumption: Only lowercase letters, no space
    # shift: Integer between 0 to 25
    # Create a dictionary. If shift=3, mapping = {'a':'d', 'b':'e'... }
    alphabets = ['a','b', … 'z']
    mapping = {}
    for i in range(26):
        letter = alphabet[i]
        replace = alphabets[ (i+shift)%26 ]
        mapping[letter] = replace

    result = "" # String to store the final result after encryption
    for letter in text[:]:
        encrypted_letter = mapping[letter]
        result += encrypted_letter

  return result
```

Encryption – Evaluation

- We used a random string of length 100,000 to evaluate the speed and efficiency of 3 functions.

- Using only strings: 0.0455 s
- Using lists: 0.0330 s
- Using dictionaries: 0.0189 s

- Using only strings (method 1) was slowest
  - We had to compute the shifts every time, even if the letter was same.
- Dictionaries (method 3) were fastest
  - As we have learnt, they are extremely fast for accessing values.
  - We computed the shifts only once, unlike using only strings.

Quiz – Multi Correct

Which of the following statements would result in an error?

A. Statement 1
B. Statement 2
C. Statement 3
D. Statement 4

```
Record = {}

L1 = [1, 2, 3]
T1 = (1, 2, 3)
A1 = "(1, 2, 3)"
S1 = {1, 2, 3}

Record[L1] = "List"   # Statement 1
Record[T1] = "Tuple"  # Statement 2
Record[A1] = "String" # Statement 3
Record[S1] = "Set"    # Statement 4
```

Quiz – Multi Correct

Which of the following statements would result in an error?

A. Statement 1
B. Statement 2
C. Statement 3
D. Statement 4

Explanation : List and Set are not hashable (as they are mutable) and thus can't be used as keys.

```
Record = {}

L1 = [1, 2, 3]
T1 = (1, 2, 3)
A1 = "(1, 2, 3)"
S1 = {1, 2, 3}

Record[L1] = "List"   # Statement 1
Record[T1] = "Tuple"  # Statement 2
Record[A1] = "String" # Statement 3
Record[S1] = "Set"    # Statement 4
```

Summary

- We have lists, sets, tuples, dictionaries
- Lists are very powerful and versatile - but some operations are slow
- Sets are limited in scope - but fast
- Dictionaries - general structure, some operations are fast, but does not allow sorting / indexing
- Chose the structure that naturally fits the needs of the problem - representing data, and operations that need to be performed
- Also consider efficiency issues for operations

Extra – Social Network

```
Friends_Book = {
    ('John','UK', 20): [('James','UK', 21), ('Mark','US', 21)],
    ('Mark','US', 21): [('John','UK', 20), ('Emily','US', 19)],
    ('Emily','US', 19): [('Mark','US', 21),('Olivia','US', 21)],
}
```

- A combination of data structures can be used to model social network graphs as shown above.
- Possible operations - Find immediate friends of a person, Check if a person exists in the network, Find 2nd degree friends of a person and much more.
- Graphs in itself is a wide topic but takeaway here is the importance of choosing the right data structure to model the graph.

Example – Student Record and CGPA

- Students take courses in a semester, get grades in them
- From the grades, the SGPA and CGPA is computed
- Student record keeps this information, along with other info about the student
- Organizing this student information:
  - Dictionary is the natural way to organize diverse data of a student
  - Within it record of courses can be kept as a list
  - Info for each course is naturally a tuple of (name, credits, grade)
- All the students in an institute can be a Dictionary of records.
- Such a structuring of data will allow for faster data access and also keep the design flexible. (can easily add a new attribute or leave some attribute value blank for a student)
- Using multiple lists or list of lists, instead of dictionary, for organising this data would pose certain limitations.

Student Record

```
Record = {

  2018001:{
      'Name': 'Aniket',
      'Age': 18,
      'Branch': 'CSE',
      'Grades':{
          "SEM1": [("MTH101",4,9),("CSE101",4,8),("ECE111",4,10),("DES101",4,10),("COM101",4,10)],
          "SEM2": [("MTH102",4,7),("CSE201",4,9),("ECE112",4,9),("CSE202",4,10),("SSH112",4,10)],
        }
    },

  2018002:{
      'Name': 'Rahul',
      'Age': 19,
      'Branch': 'ECE',
      'Grades':{
          "SEM1": [("MTH101",4,8),("CSE101",4,9),("ECE111",4,9),("DES101",4,9),("COM101",4,10)],
          "SEM2": [("MTH102",4,9),("CSE201",4,8),("ECE112",4,10),("CSE202",4,10),("SSH112",4,9)],
        }
    }
}
```

Calculating CGPA

```
def calculate_cgpa(roll_number):
    cgpa = 0
    credit = 0

    for sem in Record[roll_number]["Grades"]:
        for subject in Record[roll_number]["Grades"][sem]:
            cgpa += subject[1] * subject[2]
            credit += subject[1]

    cgpa = cgpa / credit
    return cgpa

def get_student_info(roll_number):
    if roll_number in Record:
        print("Name:", Record[roll_number]["Name"])
        print("Age:", Record[roll_number]["Age"])
        print("Branch:", Record[roll_number]["Branch"])
        print("CGPA:", calculate_cgpa(roll_number))
    else:
        print("Student not found")

num = int(input("Enter Roll Number: "))
get_student_info(num)
```