Programming for DS

Python DS vs DS vs RAM Memory

Check List

Tuple

* Immutable, ordered, indexed, allows duplicates, heterogeneous, light weight, hashable (tuple can be used as dictionary key), nestable.

info = ("Swetha", True, (10, 20)) to access tuple we use [] ← index.

empty = ()

info = ("Sasikala", 2025, True, (10,20))

print(len(info)) = 4

print(len(empty)) = 0

print(len(info[-1])) = 2.

combined = info + ("AI", "Python")

o/p: ('Sasikala, 2025, True, (10,20), AI, Python)

print('Sasikala' in info) True

name, year, bol, coords = info

print(name) → Sasikala

name, *rest = info ⟶ Is name, rest a tuple.

result = info(2::-1)

result = info[::-2]

data = [(1, 'b'), (3, 'a'), (2, 'c')]

sorted(data) ⟶ sorts by first element

sorted(data, key = lambda x : x[1]) # sorts by second ele

data = ((3, 'c'), (4, 'a')) → sort not possible bcz tuple.

data = tuple(list1) → converting list to tuple.

Q1) Create a tuple of tuples containing student names & scores. Sort tuple by scores in descending order.

Student = (('A', 80), ('B', 95), ('C', 70))

~~sorted(data, key = lambda~~

sorted_tuple = tuple(sorted(student, key = lambda x : x[1], reverse = True))

Q2) From a list of tuples (product, price) filter out products priced below 500.

Q4) Access 7 from nested = (1, (2, 3, (4, 5, (6, 7))))

nested[1][2][2][1] or nested[-1][-1][-1][-1]

Q5) & Q6) Hw

8) Fn that accepts any no args & returns them as tuple

```
def pack_into_tuple(*args):
    return args;

l=[]
for i in args:
    l.append(i)

return tuple(l)
```

9) Perform XOR (element wise) b/w 2 tuples of equal length.

- t1 = (1,2,3) t2 = (3,2,1)

o/p : (2,0,2)

```
result = tuple(a^b for a,b in zip(t1, t2))
```

18/7/25

Dictionary is a built in DS that stores key value pairs.

Characteristics: ordered, mutable, indexed by keys. keys must be unique & immutable, any data type.

Common operations:

```
dict_name[key] - access
dict_name["key"] - value  - add item
merge = d1 | d2
d={} for i in range(5):
        sq = i**2
        d[i] = sq
    } do append(x:x*2)
```

grades = {'A': 85, 'D': 92, 'K': 78}

passed = {k:v for k, v in grades.items() if v >= 80}

for k in grades:

    if grades[k] >= 80:

        passed.append(k: grades[k])

access values using index number.

value_at_index1 = list(d.values())[1]

HW

Q) Create last 3 days temp avg & store atleast 5 readings/d using dictionary (Date is key) (f/o both dict)

Ex: d = {'18.25': [30, 38, 36, 33, 35]

}

Q) Find best rating & lowest price of amazon products. Get product id, rating & price

Sets

Sets are definate, unordered

A = {0, 2, 4, 4, 6, 8}

B = {1, 2, 3, 4, 5}

Union: A|B = 0, 1, 2, 3, 4, 5, 6, 8

Intersection: A&B = 2, 4

Symmetric diff: A^B = 0, 1, 3, 4, 5, 6, 8 (present twice in A)

Frozen Set - once created cannot be changed

Multiset - allows duplicates, unordered, hashable

pip install multiset

List - ordered, mutable, allow duplicates, heterogeneous

Adding elements to list - append(), extend(), insert

Removeing elements - remove(), pop(), del

List Comprehension - concise way of creating a list

squares = [x**2 for x in range(1,6)]

Q) prices = [7,10,1,3,6,9,2]

Task is to find the maximum profit possible by buying & selling stocks on different days

A) prices = [7, 10, 1, 3, 6, 9, 2]

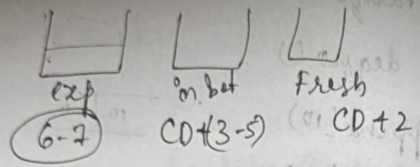
```
min_profit = prices[0]
max_profit = 0
for i in range(1, len(arr)):
    profit = arr[i] - min_profit
    min_profit = min(arr[i], min_profit)
    max_profit = max(max_profit, profit)
print(max_profit)
```

1 → 9 : 8

from collections import deque exp = [1, 5, 4, 6] 7 days.

~~Stack = deque()~~

~~if expiry < 0:~~
max_exp = max(exp)
min_exp = min(exp)
Stack.append(max_exp)
Stack.append(min_exp)



7 items

Stack1 = deque()
Stack2 = deque()
Stack3 = deque()

~~use~~ current_date = get in python.

expiry = date - current data

---

from datetime import datetime
print(datetime.today().date())

O/P: YYYY-MM-DD

25/7/25

Q1) count vowels.

```
count = 0
word = input()
vowels = 'aeiou'
word = word.lower
for char in word:
    if (char == 'a' or char == 'i' or char == 'o' or
        char == 'e' or char == 'u'):
        count += 1
print(count)
```

Q2) word = "good morning"
word.replace("morning", "evening")
print(word)

Q3) word = "safe"
sentence = "wearable device for safety"
if word in sentence:
    print("Present") # Precent

Q4) sentence = "Programming for data science"
sentence.find("data") rest?
word = "Data"
res = word.lower()

Q5) l = ['Hi', 'Welcome', 'Hello']
res = '-'.join(l)
print(res) # Hi-Welcome-Hello

Q6) sentence = "- Programming for data science -"
sentence.skip()
print(sentence) # Programming for data Science

Q7) pin = 560090
print(pin.isdigit()) # True

Q8) PAN = 'GXW2431G'
print(PAN.isalnum()) # True

2/8/25 (In class exercise)

Q) Define a Base class User with attribute name & email

```
Class User:
    def __init__(self, name, email):
        self.__name = name
        self.__email = email

    def display_info(self):
        print("Name : {self.__name}")
        print("Email : {self.__email}")

    def get_name(self):
        return self.__name
```

encapsulation

instance variables

instance variables must always be written as self.var

(Getter method)

Public method to access private variable

Internally in Python every Private variable becomes _

Ex: self.__name = name

Translates to :- self._User__name ← Name mingling

(or) self._ClassName__varName

Q) Implement Child class using super()

Create a class Student inheriting from User. Add a new attribute course & use super() to call parent initializer. Override display_info() to include course information & use super() to call parent class method display_info()

```
Class Student(User):
    def __init__(self, name, email, course):
        self.course = course
        super().__init__(name, email)

    def display_info(self):
        print("Student Info")
        print("Name : {self.get_name()}")
        print("Email : {self.__email}")
        print("Course : {self.course}")
```

calling parent initializer

accessing private var

independent class

```
Class Mentor:
    def __init__(self, specialization):
        self.specialization = specialization

    def display_info(self):
        print("Mentor info")
        print("Specialization : {self.specialization}")
```

Multiple inheritance

```
~~Class Mentor~~:
Class TeachingAssistant(Student, User):
    def __init__(self, name, email, course,
                 specialization):
        super().__init__(name, email, course)
        self.specialization = specialization
```

calling parent classes

```
def display_info(self):
    print("Teaching assistant info")
    print(f"Name : {self.get_name()}")
    print(f"Email : {self._email}")
    print(f"Course : {self.course}")
    print(f"Specialization : {self.specialization}")

def show_details(obj):
    obj.display_info()

u = User("Alice", "@gmail")
s = Student("Bob", "@yahoo", "AI")
ta = TeachingAssistant("Joe", "@hotmail", "DS", "ML")
m = Mentor("DS")

show_details(u)
show_details(s)
show_details(ta)
show_details(m)
```

} each class prints diff values for same fn

Polymorphism

O/P:

```
Alice
@gmail

Bob
@yahoo
AI

TA :
Joe
@hotmail
DS
ML

Mentor :
DS
```

display_info() ↙ Method overriding

When a child class defn a method with same name as a method in its parent class, & modifies its behavior it is called Method overriding

NLTK operations

1. Tokenization - Splitting text into parts.

* Breaking down a piece of text into smaller units (tokens) like words or sentences.

* word tokenization → split into words / punctuation

Ex: text = "Hello, my name is"
print(word_tokenize(text))

O/p: ['Hello', ',', 'my', 'name', 'is']

* Sentence tokenize → splits into sentences.

Ex: print(sent_tokenize(text))

2. Stopword Removal - Removing common filler words.

* Stopwords are very common words (like 'is', 'the', 'and') that usually dont add much meaning. Removing them helps focus on keywords.

Not in syllabus ↗

3. Stemming - Cutting words to ~~their~~ their root

* Chops off word endings to reduce them to a base form.

* Does not guarantee a real world

Ex: flies —stemming→ fli

Ex: from nltk.stem import PorterStemmer

ps = PorterStemmer() ← applies a set of rules to remove endings

print(ps.stem("running")) → run

print(ps.stem("flies")) → fli

4. Lemmatization - dictionary form.

* Converts words to their base form based on grammar & vocal
* unlike stemming, returns valid words

Ex: from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("running", pos="v"))
↳ run

5. POS Tagging - Identifying grammatical roles.

* Parts of speech tagging assigned to each word

Ex: import nltk
words = word_tokenize("NLTK is great for tasks")
print(nltk.pos_tag(words))

Ex: [('NLTK', 'NNP'), ('is', 'VBZ'), ('great', 'JJ'), ('for', 'IN'), ('tasks', 'NNS')]

6. Named Entity Recognition (NER) - Finding names, places, dates

NER detects real world named entities like people, organizations, locations.

Sentence = "Apple is buying a UK startup for 1$ billion"
Apple → Organization
UK → Detected as geo political entity
1$ billion → Money

Extra: * word_tokenize() internally calls sent_tokenize()
* sent_tokenize() relies on pre-trained sentence boundaries detection model called punkt.