Python Syllabus

DATAPLAY

- Print Statement and Input Statement
- Variables
- Data Types in Python
  - Primitive
  - Non-Primitive
    - List, Dictionary, Tuple, Set, String
- Datatype Conversion (Implicit / Explicit)
- Indexing and Slicing
- Commonly Used String Operations
- Types of Operators
  - Bitwise (&) and Logical (and) Operators
  - Equality vs Identity Operators
- Loops-> For Loop, While Loop
- Built-in Functions-> enumerate, zip, range
- Control flow statements
  - If, Else, and Elif Statements
  - Break and Continue

- Functions
  - Passing Arguments
  - Positional Arguments
  - Keyword Arguments
  - Mixing Positional and Keyword Arguments
  - Default Parameter Values
  - Variable-Length Positional Arguments
- Comprehension -> List / Dictionary Expression
- Expression -> Lambda/Generator Expression
- Python Libraries
  - Numpy
  - Pandas

    Filtering Pandas DataFrame (loc vs iloc)

    Multi Index DataFrame

    Concatenation /Merge

    Data Sorting

    Data Cleaning

    Data Transformation
- **Industry Case Studies**

DATAPLAY

What we will cover in the video:

- Anaconda installation
- Intro To Python
- Python Expression
- Python Variables
- Print and Input Statement

High Level vs Low Level language

DATAPLAY

Feature	High-Level Language	Low-Level Language

Definition	High-level languages are designed for humans, using keywords similar to English for easier reading and writing.
	
	Addition of two numbers in Python:
	a=5
	b=5
	a+b
	output: 10
		Low-level languages directly interact with hardware, using complex instructions like machine code or assembly language.
		
		Addition of two numbers in Assembly:
		MOV AX, 5
		MOV BX, 5
		ADD AX, BX
		output: 10

Abstraction	High abstraction	Low abstraction
Focus	Programmer productivity	Hardware control
Readability	Easier to read	Harder to read
Portability	More portable	Less portable
Development Time	Faster development	Slower development
Examples	Python, Java, C++	Assembly, Machine code

Expression

DATAPLAY

An expression is a fundamental building block of your code. It's a combination of values, variables, operators, and function calls that evaluates to a single result.

Components of an Expression:

- **Values:** These can be numbers (integers, floats), strings of text, booleans (True or False), complex values or special values like `None`.
- **Variables:** Variables act as named containers that store data during program execution. You can use variable names to refer to their stored values within expressions.
- **Operators:** Operators perform operations on values. Python offers various operators for arithmetic calculations (e.g., `+`, `-`, `*`, `/`), comparisons (e.g., `==`, `!=`, `<`, `>`), logical operations (e.g., `and`, `or`, `not`), and more.
- **Function Calls:** Functions are **reusable blocks** of code that perform specific tasks. You can call functions within expressions to utilize their functionality and incorporate their return values into the overall expression result.

Types of Expression

DATAPLAY

Assignment Expression	x = x+3 -> x+=3
Arithmetic Expression	2+3, 2*3, 2**3, 7/2, 7//2 (floor division), 7%2(modulus)
Relational Expression	12 ==2 , 12 != 12 , 12 > 12, 12<2
Logical Expression	(x>0) and (y>0) , (x>0) or (y>0) , not (x>0)
Bitwise Expression	5 & 3, 5 | 3, 5^3, ~5
Membership Expression	3 in [1,2,3], 4 not in [1,2,3]
Identity Expression	x = 5, y = 5 -> x is y *Check address of x and y
Equivalent Expression	x = 5, y = 5 -> x == y *Check value of x and y
Conditional Ternary Expression	condition ? value_if_true : value_if_false
Lambda Expressions	To be studied later..
Generator Expression	To be studied later..

DATAPLAY

Difference between Bitwise (&) and Logical (and)

Feature	&& / and (Logical AND)	& (Bitwise AND)
Purpose	Conditional logic	Bit manipulation
Operates on	Boolean expressions	Integers
Output	Boolean (True/False)	Integer (modified bits)
Short-circuit evaluation	Yes (stops at False)	No
Example	x > 0 and y < 10	flag_byte & 0b00000011

Variables in Python

DATAPLAY

- **Definition**

  Variables act as named containers that store data throughout your program's execution. They are used to keep track of values that can be used and manipulated throughout a program.

- **Creating Variables :**

  Variables are created simply by assigning a value to a name which is case-sensitive (e.g., age and Age are different variables.

- **Naming convention -> Camel Case**

  Start with a letter or underscore (_).
  Can contain letters, numbers, and underscores.
  The first word starts lowercase, and subsequent words have their first letter capitalized (e.g., weddingGuestList, totalCost).

- **Data Types:**

  Python is dynamically typed, meaning you don't need to declare the data type beforehand. The data type is inferred based on the assigned value.

- **Mutable (Reassignment) / Immutable:**

  Variables themselves are not inherently mutable or immutable. Instead, it's the objects that the variables reference that are mutable or immutable. E.g. Bank Transactions requirement

- **Scope:**
  Scope determines their accessibility within different parts of your code.
  -> Local variables are defined within functions and are only accessible within those functions.

  -> Global variables are defined outside functions and can be accessed from anywhere in the program (use with caution).

- **Here are some additional points to remember:**
  Avoid using reserved keywords (like if, for, while) as variable names.
  Use meaningful variable names to improve code readability.
  Practice good variable naming conventions for better code organization.

- **Variable Deletion:** In Python, you can delete variables using the del keyword. We can talk about when and how to use this feature.

How to get address of variable in Python

```
# Create a variable to store the number of wedding guests
numWeddingGuests = 100
id(numWeddingGuests)
```

```
140723418144648
```

Print Statement

DATAPLAY

- The **print()** statement in Python is used to output text to the console or to a file.
- **Basic Usage:**
  - print("Hello, world!")
- **Printing Variable :**
  - name = "Mahima"
  - age = 25
  - print(f"Name: {name}, Age: {age}") # Using f-strings for formatted output
- **Newline Character**:
  - By default, the print statement adds a newline character (\n) at the end, moving the cursor to the next line.

Input Statement

- In Python, the **input()** function is used to get input from the user.
- It allows the user to enter data from the keyboard while the program is running.
- By default, input always returns the user's input as a string, even if they enter a number.
- num_guests = int(input("How many guests are attending? "))

DATAPLAY

What we covered in the last video:

- Anaconda installation
- Intro To Python
- Python Expression
- Python Variables
- Print and Input Statement

Today we will look into Python Data Types

DATAPLAY

Data Types in Python –>

Primitive Data Types

DATAPLAY

Data Type	Description	Examples	Space Allocated in Memory
int	A whole number (integer)	10, -5, 0, 10000	4 bytes
float	A decimal number (floating point)	3.14159, 2.71828, -0.5, 1.0	8 bytes
complex	A number with both real and imaginary parts	3 + 4j, -2.5 + 1.3j, 1 - 2.5j	16 bytes
bool	A binary value indicating true or false	True, False	1 byte
NoneType	A special value representing nothing or absence of a value	**None**	The exact memory allocation can vary, but it's likely a small amount (e.g., 4-8 bytes).

None properties:

- There's only one instance of **None** that exists throughout your entire Python program.
- This makes **None** behave similarly to a singleton (one object of its kind)

Non Primitive- Mixed Data Types

DATAPLAY

Data Type	Examples	Properties
List	tasks = [flowers, decorations, food, guest invitations]	Mutable Ordered Heterogeneous Duplicate Allowed
Tuple	bridesmaid_role= (“Sarika”,”Ritika”)	Immutable Ordered Heterogeneous Duplicate Allowed
Dictionary	guest_preference = {“Ram": “South Indian“, “Hari”: “North-Indian”}	Mutable Unordered Heterogeneous Duplicate key not allowed
Set	guest_set = {“Sarika“, “Divya“,”Kavita”}.	Mutable Unordered Heterogeneous Duplicate Not Allowed
String	my_str = “#WeddingLock”	Immutable Ordered Homogeneous

Non Primitive Data Types

Wedding Season

DATAPLAY

Once upon a time, Emma and Jack planned their dream wedding.

Created a **list** of songs to be played during different parts of the wedding ceremony.
Songs list: ["Here Comes the Bride", "Canon in D", "A Thousand Years", "Here Comes the Bride", "Marry You"]

Their **set** ensure no duplicate guests were invited, like when they included only unique names such as {“Nishant“ ,“Mahima“}.

Special roles like bridesmaid_role (“Sarika”, “Ritika”) and groomsman_role (“Ram”, “Shyam”) were assigned using **tuples**, each role being fixed and unique.

Details for each vendor for ready reference {“flowers ": “897441322“, “food”: “997442324”} such as dietary preferences were stored in a **dictionary**.

Wedding Card Invitation stored as **string** “#WedLock!”

Non-primitive data types can exhibit **infinitely nested structures,** accommodating **mixed data types** at each level

DATAPLAY

```
nested_list = [1, "hello", [3.14, True], {"key": "value"}]
nested_list
```

```
[1, 'hello', [3.14, True], {'key': 'value'}]
```

```
nested_dict = {
    "name": "John",
    "age": 30,
    "address": {"street": "123 Main St", "city": "Anytown", "zipcode": 12345},
    "contacts": ["email@example.com", "555-1234"],
}
nested_dict
```

```
{'name': 'John',
 'age': 30,
 'address': {'street': '123 Main St', 'city': 'Anytown', 'zipcode': 12345},
 'contacts': ['email@example.com', '555-1234']}
```

DATAPLAY

DataType	Creation	Accessing via index	Slicing
List	empty_list = []	my_list[index]	my_list[start:end:step]
Tuple	empty_tuple = ()	my_tuple[index]	my_tuple[index1:index2]
Dictionary	empty_dict = {}	Not Applicable (Not Ordered)	Not Applicable (Not Ordered)
Set	empty_set = {}	Not Applicable (Not Ordered)	Not Applicable (Not Ordered)
String	empty_string = “”	my_string[index]	my_string[index1:index2]

DATAPLAY

Indexing And Slicing

Indexing

- **Zero Hero:** Indexing starts at 0, not 1! First item = index 0, second item = index 1.
- **Time Travel?** Go backwards! Negative indexes let you access elements from the end.
- **Unlock Elements:** Use indexing to grab specific items from lists, strings, and other ordered data.
- **Square Brackets are your Key:** Use [] with the index to unlock the element you want.

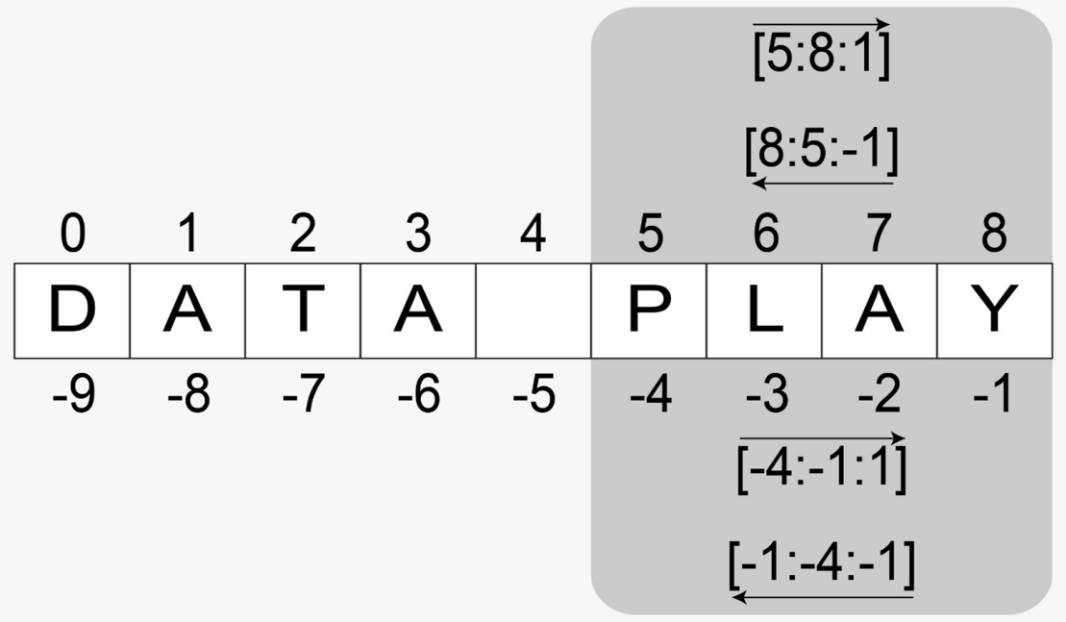
Example:

```
my_list = [1, 2, 3, 4, 5]
first_element = my_list[0]      # It will give element at 0th index that is 1
last_element = my_list[-1]      # It will give element at last  index that is 5
```

Slicing Mastery with my_list[start:end:step]

DATAPLAY

- **Grab a Chunk:** As you mentioned, slicing lets you target a specific section of your list.
- **Start (Optional):** Defaults -> 0, Inclusive.
- **End (Optional):** Defaults -> Last Element of list, Exclusive.
- **Step It Up (Optional):** Default -> 1
  +ve step value -> move forward
  -ve step value -> move backward



Helps with list reversal.

DATAPLAY

DataType	Add Element in original variable		Removing element from original variable	
	Default Position	**At Particular Index**	**By value**	**By index**
List	my_list.append(2)	my_list.insert(index, value)	my_list.remove('egg')	my_list.pop(1)
Tuple	N/A (Immutable)	N/A (Immutable)	N/A (Immutable)	N/A (Immutable)
Dictionary	my_dict[key] = value	N/A (Not Ordered)	del my_dict['key']	N/A (Not Ordered)
Set	my_set.add(1)	N/A (Not Ordered)	my_set.remove(value)	N/A (Not Ordered)

Tuple is immutable then how can we modify it ?

-> By assigning a new variable

DATAPLAY

DataType	Modify an element in original variable	Clearing all elements	Find length	Count of Occurrence of an element
List	my_list[index] = value	my_list.clear()	len(my_list)	my_list.count(value)
Tuple	N/A (Immutable)	N/A (Immutable)	len(my_tuple)	my_tuple.count(value)
Dictionary	my_dict['key’] = value	my_dict.clear()	len(my_dict)	N/A
Set	N/A ()	my_set.clear()	len(my_set)	N/A
String	N/A (Immutable)	N/A (Immutable)	len(my_string)	my_str.count(value)

DATAPLAY

DataType	Extend (Modifies Original object)	Concatenate (Creation of a new object)
List	my_list.extend(other_list)	new_list = my_list + other_list
Tuple	N/A (Immutable)	new_tuple = my_tuple + other_tuple
Dictionary	my_dict.update(other_dict)	new_dict = {**my_dict, **other_dict}
Set	my_set.update(other_set)	new_set = my_set.union(other_set)
String	N/A (Immutable)	new_string = my_string + other_string +”.” new_string = f"{my_string} {other_string}.“

DATAPLAY

DataType	Checking Existence/Membership	Sorting (In Place)	Sorting (Return new variable)
List	value in my_list	my_list.sort()	sorted(my_list)
Tuple	value in my_tuple	N/A (Immutable)	sorted(my_tuple)
Dictionary	key in my_dict.keys() value in my_dict.values()	N/A (Not Ordered)	N/A (Not Ordered)
Set	value in my_set	N/A (Not Ordered)	N/A (Not Ordered)
String	value in my_str	N/A (Immutable)	sorted(my_string)

Set Operations

DATAPLAY

Operation	Description	Example
Intersection	Create a new set with the common elements between two or more sets	new_set = my_set.intersection(other_set)
Difference	Create a new set with the elements in one set but not the other	new_set = my_set.difference(other_set)
Symmetric Difference	Create a new set with the elements in either set but not both	new_set = my_set.symmetric_difference(other_set)
Subset	Check if one set is a subset of another set	if my_set.issubset(other_set):
Superset	Check if one set is a superset of another set	if my_set.issuperset(other_set):

Data Type Conversion (Typecasting)

DATAPLAY

Feature	Implicit Conversion	Explicit Conversion (Type Casting)
Triggered by	Compiler automatically	Programmer manually
Risk of data loss	No (safe conversions only)	Yes (if conversion is not possible)
Direction	Smaller type to larger type	Any type to any type
Use case	Simplify code, maintain readability	Force conversion, handle specific data types
Example	# Adding an integer and a float num1 = 10, num2 = 2.5 num1 + num2 output: 12.5	# Converting a string to an integer num_str = "10" +10 int(num_str) output: 10

DATAPLAY

Control Flow Statements –>

DATAPLAY

if, else, and elif Statements

```python
# Example: Checking if a number is even or odd using if...else statement
number = 10
if number % 2 == 0:  # Checking if the number is even
    print(f"{number} is even.")
else:  # Executed if the number is odd
    print(f"{number} is odd.")
```

```
10 is even.
```

Why not multiple if ????

DATAPLAY

```
# Example: Checking the range of a number using if, elif, else statements
number = 25
if number < 0:
    print("The number is negative.")
elif number == 0:
    print("The number is zero.")
elif 0 < number < 20:
    print("The number is between 1 and 19.")
elif 20 <= number < 50:
    print("The number is between 20 and 49.")
else:
    print("The number is 50 or greater.")
```

```
The number is between 20 and 49.
```

Loops

DATAPLAY

Feature	for Loop	while Loop
Use Case	Used for iterating over a known sequence or range.	Used when the number of iterations is not known in advance.
Syntax	for variable in iterable:	while condition:
Iteration Control	Iterates until it exhausts the items in iterable.	Iterates until the specified condition becomes false.
Initialization	No explicit initialization is needed.	Requires an explicit initialization of the control variable before the loop.

Example	`# Iterate over elements in a list`
	`numbers = [1, 2, 3]`
	`for num in numbers:`
	`  print(num)`
	
	1
	2
	3
		`# Print numbers from 1 to 3 using a while loop`
		`num = 1`
		`while num <= 3:`
		`  print(num)`
		`  num += 1`
		
		1
		2
		3

For Loop for various DS

DATAPLAY

```python
# For Loop for List:
wedding_tasks = ["decorations", "venue booking", "food", "guest invitations"]
for task in wedding_tasks:
    print(task)

# For Loop for Tuple:
bridesmaid_role = ("Sarika", "Ritika")
for bridesmaid in bridesmaid_role:
    print(bridesmaid)

# For Loop for Set:
guest_set = {"Sarika", "Divya", "Kavita"}
for guest in guest_set:
    print(guest)

# For Loop for Dictionary (Key-Value Pairs):
vendor_details = {"flowers": "1240-3343345", "wedding_rings": "4787-5636"}
for key, value in vendor_details.items():
    print(key, value)

# For Loop for String:
wedding_hashtag_1 = "#TheNotebookLoveStory"
for char in wedding_hashtag_1:
    print(char)
```

Break and Continue Statements

DATAPLAY

Break Statement:

- The **break** statement is used to terminate a loop prematurely. When the **interpreter** encounters a **break** statement in a loop, it immediately exits the loop and continues with the next statement outside the loop.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using 'break' to terminate the loop when a condition is met
for number in numbers:
    if number == 5:
        print("Number 5 found. Exiting the loop.")
        break  # Exits the loop
    print(number)
```

```
1
2
3
4
Number 5 found. Exiting the loop.
```

Continue Statement

DATAPLAY

- The **continue** statement is used to skip over the current iteration of a loop and move on to the next iteration. When the **interpreter** encounters a **continue** statement in a loop, it skips the rest of the code in the current iteration and moves on to the next iteration.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using 'continue' to skip printing even numbers
for number in numbers:
    if number % 2 == 0:
        print(f"Skipping even number: {number}")
        continue  # Skips even numbers
    print(number)
```

```
1
Skipping even number: 2
3
Skipping even number: 4
5
Skipping even number: 6
7
Skipping even number: 8
9
Skipping even number: 10
```

DATAPLAY

- Pattern Printing
- Array Question (Easy and Medium)

Built In Function

DATAPLAY

Iterator: An object representing a stream of data.

enumerate()

```
for index, index in enumerate(sequence):
    #code block to execute
```

The enumerate() function is used to iterate over a sequence while keeping track of the index of each item.

zip()

```
for item1, item2, ... in zip(iterable1,iterable2,...):
     #code block to execute
```

The zip() function is used to combine multiple iterables element-wise into tuples.

range

```
for variable in range(start, stop , step):
    #code block to execute
```

The range() function is used to generate a sequence of numbers.

enumerate()

DATAPLAY

- **Purpose:** Adds a counter with an index to each item in an iterable.
- **Syntax:** enumerate(iterable, start=0)
- **Parameters:**
  iterable: Any iterable object (list, tuple, string, etc.).
  start (optional): Starting index for the counter (defaults to 0).
- **Behavior:** Returns an iterator of tuples, where each tuple contains the index and the corresponding item from the iterable.

```python
# Basic counting on a list
fruits = ["apple", "banana", "cherry"]
for i, fruit in enumerate(fruits):
    print(f"Index {i}: {fruit}")
```

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

```python
# Basic counting on a list
fruits = ["apple", "banana", "cherry"]
for i, fruit in enumerate(fruits, start=1):
    print(f"Index {i}: {fruit}")
```

```
Index 1: apple
Index 2: banana
Index 3: cherry
```

Continued…..

DATAPLAY

Enumerate Over Tuple ->

```
coordinates = (5, 3)
for i, coordinate in enumerate(coordinates):
    print(f"Index {i}: {coordinate}")
```

```
Index 0: 5
Index 1: 3
```

Zip()

DATAPLAY

Zipping Up Your Data: This emphasizes the function's ability to combine elements from same/different iterable types.

The Parallel Packer: This highlights the ability of zip to process elements from multiple iterables in parallel.

Tuples on the Fly: This focuses on the output of zip, which is an iterator of tuples created from the input iterables. Ex. tuple(zip([1,2,3,4,5],['a','b','c'])) = ((1,'a'),(2,'b'),(3,'c'))

Combines Same and different length Iterables

```python
# Zipping numbers and letters
numbers = [1, 2, 3]
letters = ["a", "b", "c"]
for num, letter in zip(numbers, letters):
    print(f"({num}, {letter})")
```

```
(1, a)
(2, b)
(3, c)
```

```python
# Using zip() with more than two iterables
numbers = [1, 2, 3]
letters = ["a", "b", "c"]
colors = ["red", "green"]
for num, letter, color in zip(numbers, letters, colors):
    print(f"({num}, {letter}, {color})")
```

```
(1, a, red)
(2, b, green)
```

zip is not recommended for unordered collections.

DATAPLAY

Zip Over different iterable types ->

```
for x,y in zip([1,2,3,4,5],('a','b','c')):
    print(x,y)
```

```
1 a
2 b
3 c
```

It is not advisable to use zip on unordered collections but in case if you use it with dictionary it will take insertion order by default

```
for a,b,c in zip([1,2,3,4,5],('a','b','c'),{'a':5,'b':2}):
    print(a,b,c)
```

```
1 a a
2 b b
```

Range()

DATAPLAY

- **Purpose:** Generates a sequence of numbers within a specified range.
- **Syntax:** range(start, stop, step=1)
- **Parameters:**
  - start: Starting value (inclusive, defaults to 0).
  - stop: End value (exclusive).
  - step (optional): Increment between values (defaults to 1).
- **Behavior:** Returns an immutable range object, which you can convert to a list using list().

```
# Custom start, stop, and step
for i in range(2, 8, 2):
    print(i)
```

```
2
4
6
```

```
# Using range() to create indices
my_list = ["hello", "world"]
for i in range(len(my_list)):
    print(f"Index {i}: {my_list[i]}")
```

```
Index 0: hello
Index 1: world
```

```
my_tuple = (1, 2, 3)
for i in range(len(my_tuple)):
    print(my_tuple[i])
```

```
1
2
3
```

String

DATAPLAY

Looping Through Each Character

```
my_string = "Hello, World!"

for char in my_string:
    print(char)
```

Looping through Indices

```
my_string = "Hello, World!"

for index, char in enumerate(my_string):
    print(f"Index: {index}, Character: {char}")
```

Looping by using range

```
my_string = "Hello, World!"

for index in range(len(my_string)):
    char = my_string[index]
    print(f"Index: {index}, Character: {char}")
```

Find transpose of a matrix using for loop

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

harrypotter_database

- Calculate the average grade of a student across all courses.
- How would you find the student with the highest average grade in each house?
- Calculate the overall average grade for each course across all houses.
- Compare the average grade distribution between different age groups (16 vs 17 year olds)
- For each house, find the subject with the biggest gap between the highest and lowest average grades.

student_dataset

- Find all courses taught by a specific teacher (e.g., Mr. Brown)
- Find all students taking a specific course (e.g., Math)
- Find the most common course taken by students and the total number of students enrolled in it.
- For each department, list all the teachers and the number of different courses they teach.

DATAPLAY

Text Given:

The round, round moon hung heavy in the night, night sky. Crickets chirped, chirped their monotonous song, a quiet, quiet symphony for sleepy ears. A fluffy, fluffy cat stalked a tiny, tiny mouse through the tall, tall grass. The cat, cat pounced, but the clever, clever mouse darted away, leaving the frustrated, frustrated feline behind. The wind whispered, whispered secrets through the leaves, leaves of the ancient oak tree. Sleep, sleep, came softly, and the world drifted off to peaceful, peaceful dreams.

Questions:

1. **Text Preprocessing:**
   - Remove comma and punctuation from text.
   - Split text into words.
2. **Unique Words**
3. **Word Lengths**
4. **Longest Word**
5. **Generating Bigrams store them as list of tuple**
6. **Dictionary with bigrams and there frequency**

Functions

DATAPLAY

Functions in Python are a way of **organizing** and **reusing** code. They allow you to write a block of code that can be called **multiple times** from different parts of your program, without having to repeat the same code over and over.

Why we should use Function:

- **Modularity**: Functions break down complex programs into smaller, more **manageable** parts. This makes your code easier to read, understand, and maintain.
- **Reuse**: Functions allow you to reuse code across your program. Once you've written a function, you can call it from **different parts** of your program without having to rewrite the same code.
- **Abstraction**: Functions can abstract away the details of how a piece of code works. For example, if you have a function that **calculates the average** of a list of numbers, you don't need to know how the function calculates the average. You just need to know what inputs it takes and what output it returns.
- **Readability**: Functions can make your code more readable by giving meaningful names to **blocks of code**. This makes it easier for other developers to understand your code.
- Functions in Python are first-class objects, which means they can be assigned to variables, passed as arguments to other functions, and returned as values from other functions which also means that they occupy memory like any other object. In Python, memory for a function is allocated at the time of definition, not at the time of call. This function object contains the code of the function, its name, default arguments (if any), and any other metadata associated with the function.

```python
def function_name(x: int, y: float) -> str:
    """
    Perform a complex computation.

    This function takes an integer 'x' and a floating-point number 'y'
    as input and returns a string result.

    Parameters:
    - x (int): An integer input.
    - y (float): A floating-point input.

    Returns:
    - str: A string result based on the computation.

    Examples:
    >>> function_name(5, 3.0)
    'Result: 8.0'
    """
    # Function Body - code block

    # Placeholder comment for some operation with parameters
    result = x + y  # Replace with the actual complex operation

    # Return statement (optional)
    return f"Result: {result}"

# Test the function
print(function_name(5, 3.0))
```

```
Result: 8.0
```

DATAPLAY

- **def** keyword defines a function.
- **function_name** is the function's name, following variable naming rules.
- **parameters** enclosed in parentheses (), are optional variables passed to the function. E.g. generate_random_num()
- **Docstring** enclosed in triple quotes """ """, describes what the function does.
- **return** statement is optional and if used passes a value back from the function.
- The function body, indented, contains the code defining its functionality.

Passing Arguments

DATAPLAY

Positional Arguments

```
def greet(name, age):
    print(f"Hello {name}! You are {age} years old.")

# Calling the function with positional arguments
greet("Alice", 30)
```

Positional arguments are passed to a function in the order in which the parameters are defined. The function interprets these arguments based on their positions.

Keyword Arguments

```
def greet(name, age):
    print(f"Hello {name}! You are {age} years old.")

# Calling the function with keyword arguments
greet(age=30, name="Alice")
```

```
Hello Alice! You are 30 years old.
```

Keyword arguments are associated with parameter names explicitly during the function call. This allows you to change the order of the parameters, especially when a function has multiple parameters, making the code more readable.

Mixing Positional and Keyword Arguments

DATAPLAY

```
def greet(name, age):
    print(f"Hello {name}! You are {age} years old.")

# Mixing positional and keyword arguments
greet("Alice", age=80)
```

```
Hello Alice! You are 80 years old.
```

You can also mix positional and keyword arguments, but the **positional arguments must come before keyword arguments to avoid ambiguity and follow syntax**.

Default Parameter Values

```
def greet(name, age=25):
    print(f"Hello {name}! You are {age} years old.")

# Using default parameter value
greet("Bob")
# Output: Hello Bob! You are 25 years old.

greet("Charlie", 40)
# Output: Hello Charlie! You are 40 years old.
```

Python also allows setting default values for function parameters. When a default value is provided for a parameter, that parameter becomes optional during the function call.

DATAPLAY

Comprehension or Expression

	List / Dictionary / Set Comprehension	Lambda Expression
Syntax	[expression for item in iterable]	lambda arguments: expression (e.g. lambda x,y : x+y) eg: add_two = lambda x: x + 2 Function call : add_two(num)
Description	Concise way to create a new collection (list, set, dictionary) based on an existing iterable.	Anonymous function defined in a single line. Useful for short, throwaway functions.
Memory Usage	Can be less memory efficient for large datasets as entire list is created at once.	Negligible memory usage for the function itself.
Use Case	General purpose for creating new collections.	Short, throwaway functions within other functions.
Example	squared_num_list = [x**2 for x in num_list]	sorted_names = sorted(names, key=lambda x: len(x))

List Comprehension

DATAPLAY

```python
# Given a list of numbers square them and store in a new list.

numbers = [1, 2, 3, 4, 5]
squared_numbers = []
for num in numbers:
    squared_numbers.append(num**2)
```

```python
squared_numbers = [num**2 for num in numbers]
```

```python
# Given a list of numbers square even numbers and store in a new list.

numbers = [1, 2, 3, 4, 5]
even_squared_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_squared_numbers.append(num**2)
```

```python
even_squared_numbers = [num**2 for num in numbers if num % 2 == 0]
```

DATAPLAY

```python
# Given a list of numbers square even numbers and square root odd
# numbers and store in a new list.

result = []
for num in numbers:
    if num % 2 == 0:
        result.append(num**2)
    else:
        result.append(num**0.5)
```

```python
result = [num**2 if num % 2 == 0 else num**0.5 for num in numbers]
```

Note : else is mandatory after if statement in list comprehension.

e.g. [num**2 if num%2==0 for num in numbers] gives syntax error.

Do List comprehension support elif ?

# Example using if, elif, and else conditions
numbers = [1, 2, 3, 4, 5, 6]

# List comprehension with nested ternary conditional expressions
result = ['even' if num % 2 == 0 else ('odd' if num % 3 == 0 else 'neither') for num in numbers]

In this example:

- If the number is even, it will be labeled as 'even'.
- If the number is odd and divisible by 3, it will be labeled as 'odd'.
- Otherwise, it will be labeled as 'neither'.

DATAPLAY

Advantages of List comprehension

- **Readability**: They make code concise and expressive, enhancing readability, especially for simple transformations or filtering.

- **Conciseness**: They replace multiple lines of code with a single line, reducing codebase size and making it more compact.

- **Performance**: They can be faster than equivalent for loops, especially for large datasets, due to optimization by the Python interpreter. Python's interpreter optimizes list comprehensions to make them efficient in terms of both time and memory usage. However, it's essential to be mindful of the complexity of list comprehensions, as overly complex expressions can reduce readability and maintainability of the code.

DATAPLAY

Are List comprehensions inherently faster than for loops ?

List comprehension speed benefits

- **Preallocation:** Memory is reserved for the entire list upfront, avoiding frequent resizing during creation.
- **Caching:** Local variables are cached for faster access within the comprehension.
- **Efficient Appending:** A specialized bytecode instruction **LIST_APPEND** efficiently adds elements.
- **Reduced Overhead:** Less overhead compared to for loops with append() calls.

Conciseness and Readability

Dictionary Comprehension

DATAPLAY

```
squares_dict = {}
for num in numbers:
    squares_dict[num] = num**2
```

```
squares_dict = {num: num**2 for num in numbers}
```

```
{key_expression: value_expression for item in iterable if condition}
```

```
{
    key_expression: value_expression if condition else alternative_value_expression
    for item in iterable
}
```

Homework

DATAPLAY

```
[2022-01-01 08:15:30] 123 login success viewer
[2022-01-01 08:20:45] 123 logout viewer
[2022-01-01 09:05:12] 123 view_page article_1 viewer
[2022-01-01 09:10:30] 789 login success admin,editor
[2022-01-01 09:30:00] 456 edit_profile editor
[2022-01-01 10:00:00] 456 view_page article_2 editor
[2022-01-01 10:30:00] 789 view_page article_1 admin
[2022-01-01 10:45:22] 123 logout viewer
[2022-01-01 11:05:30] 789 logout admin
[2022-01-01 11:15:00] 456 view_page article_2 editor
[2022-01-01 11:20:45] 123 login success viewer
[2022-01-01 12:30:00] 456 logout editor
[2022-01-01 12:45:22] 789 edit_profile admin
[2022-01-01 13:15:30] 123 view_page article_3 viewer
[2022-01-01 13:30:45] 789 view_page article_4 admin
[2022-01-01 14:10:00] 123 logout viewer
[2022-01-01 14:20:00] 123 login success admin
[2022-01-01 15:00:00] 123 view_page article_4 admin
[2022-01-01 15:30:00] 789 view_page article_5 admin
[2022-01-01 16:00:00] 123 logout admin
[2022-01-01 16:30:00] 456 edit_profile editor
[2022-01-01 17:00:00] 789 edit_profile admin
[2022-01-01 18:00:00] 456 logout editor
[2022-01-01 18:15:00] 789 logout admin
```

1. **User Activity Count:**

    1. How many activities did each user perform in the log data?
    2. Can you provide a breakdown of the activities for user ID 123?

2. Action Type Frequency:-> What is the frequency of each action type across all users in the log data?

3. Session Analysis: -> List the users along with the number of sessions each user had?

4. Most Frequent Action in Each Session:- > For each user session, what was the most frequently performed action type?

Customer Account Management

DATAPLAY

I. Customer Management
create_customer(customer_id: str, name: str) -> str

II. Account Management
create_account(customer_id: str, account_type: str, initial_balance: float = 0) -> Dict[str, Any]

III. Transaction Processing
deposit(account: Dict[str, Any], amount: float) -> None
withdraw(account: Dict[str, Any], amount: float) -> bool

IV. Account Information
display_account_info(account: Dict[str, Any]) -> None

DATAPLAY

User Activity Count:

- **456:** 5 activities
- **789:** 9 activities
- **123:** 9 activities

Breakdown for User ID 123:

- login success (viewer): 2
- login success (admin): 1
- logout (viewer): 3
- view_page article_1 (viewer): 1
- view_page article_3 (viewer): 1
- view_page article_4 (admin): 1

Action Type Frequency:

- view_page: 8
- logout: 9
- login success: 6
- edit_profile: 4

User 123:

- **Session 1 (viewer):** 2022-01-01 08:15:30 - 2022-01-01 08:20:45 (Most frequent action: logout)
- **Session 2 (viewer):** 2022-01-01 09:05:12 - 2022-01-01 10:45:22 (Most frequent action: view_page)
- **Session 3 (viewer):** 2022-01-01 11:20:45 - 2022-01-01 14:10:00 (Most frequent action: view_page)
- **Session 4 (admin):** 2022-01-01 14:20:00 - 2022-01-01 16:00:00 (Most frequent action: view_page)

User 789:

- **Session 1 (admin):** 2022-01-01 09:10:30 - 2022-01-01 10:30:00 (Most frequent action: view_page)
- **Session 2 (admin):** 2022-01-01 11:05:30 - 2022-01-01 11:05:30 (Most frequent action: logout)
- **Session 3 (admin):** 2022-01-01 12:45:22 - 2022-01-01 13:30:45 (Most frequent action: logout)
- **Session 4 (admin):** 2022-01-01 15:30:00 - 2022-01-01 15:30:00 (Most frequent action: logout)
- **Session 5 (admin):** 2022-01-01 17:00:00 - 2022-01-01 18:15:00 (Most frequent action: logout)

User 456:

- **Session 1 (editor):** 2022-01-01 09:30:00 - 2022-01-01 10:00:00 (Most frequent action: view_page)
- **Session 2 (editor):** 2022-01-01 11:15:00 - 2022-01-01 12:30:00 (Most frequent action: view_page)
- **Session 3 (editor):** 2022-01-01 16:30:00 - 2022-01-01 18:00:00 (Most frequent action: edit_profile)