Lecture 1: Introduction to Python language

What is a Language?

What is a Programming language

1. A **programming language** is a formal system of instructions used to communicate with a computer.
2. Enables developers to write programs for various tasks such as web development, data analysis, and automation.
3. **Examples**: C, Java, Python, JavaScript.

Why Python

1. **Ease of Use**: Simple syntax, ideal for beginners.
2. **Versatility**: Supports multiple programming paradigms (object-oriented, procedural, functional).
3. **Extensive Libraries**: Rich ecosystem for tasks like data analysis, machine learning, and web development.
4. **Community Support**: Large and active community for troubleshooting and learning.
5. **Interpreted Language**: Python is executed line by line, which makes debugging easier.
6. **Cross-Platform**: Python programs can run on various platforms without modification (Windows, macOS, Linux).
7. **Scalability:** While often seen as a scripting language, Python scales well for larger applications when combined with appropriate frameworks or tools (e.g., Django)

What is Python

1. **Python** is a high-level, interpreted programming language known for its simplicity and readability. It was created by **Guido van Rossum** in the late 1980s and first released in **1991**. Python emphasizes **code readability** and supports multiple programming paradigms, including **object-oriented**, **procedural**, and **functional programming**.
    a. Monty Python's Flying Circus
2. Older than Java (1995).

Where Python is Used?

1. **Web Development**: Frameworks: Django, Flask.

2. **Data Science and Machine Learning**: Predictive modeling and data analysis using NumPy, Pandas, TensorFlow, scikit-learn.
3. **Automation (Scripting)**: Automating repetitive tasks.
4. **Game Development**: Framework: PyGame.
5. **Scientific Computing**: Libraries: SciPy, Matplotlib.
6. **Artificial Intelligence (AI) and Deep Learning**: Libraries: Keras, PyTorch.
7. **Web Scraping**: Libraries: BeautifulSoup, Scrapy.
8. **Cybersecurity**: Used for creating tools to test system vulnerabilities.

Companies using Python: Google**,** Netflix, Instagram, Spotify, Dropbox, NASA, Uber, Reddit, IBM, 10000 coders dashboard

Some disadvantages of Python:

1. Slower Execution Speed
    - Interpreted language, making it slower than compiled languages like C++ or Java.
    - Not ideal for performance-critical applications (e.g., real-time systems, gaming engines).
2. Memory Consumption
    - Python's dynamic typing and garbage collection can lead to higher memory usage.
    - Not suitable for applications where memory optimization is crucial.
3. Global Interpreter Lock (GIL)
    - The GIL restricts the execution of multiple threads at the same time.
    - This makes Python less efficient for multithreaded applications, especially on multi-core processors.
4. Runtime Errors
    - Python's dynamic nature allows runtime errors to occur if code is not thoroughly tested.
    - This can lead to bugs in production if not handled properly.
5. Limited Support for Low-Level Programming
    - Languages like C or Rust are better suited for such tasks.

Python Installation:

1. Download link: Download Python | Python.org
2. Commands to check installations:
    a. In cmd: python --version

Different Modes of Working: Python provides various modes to write and execute code, each suited for different workflows and purposes.

1. **Interactive Mode:** Python code is executed directly in the Python shell or terminal.

    a. In cmd: type py or python to enter into the python shell.
        i. **exit()** to come out of the python shell.
    b. Search idle in the window's search to directly enter the python shell.
2. **Script Mode:**
    a. Write Python programs in a .py file and execute them as a whole.
    b. Run using the command: python filename.py.
3. **Integrated Development Environments (IDEs):**
    a. Use tools like PyCharm, Jupyter Notebook, or VS Code for an enhanced coding experience.
    b. Offers features like debugging, syntax highlighting, and autocompletion.

Python Fundamentals:

1. **Variables**
    a. A variable is a container. Called a variable because we can change the value of a variable.
    b. Snake Case for variables (using _'s), Pascalcase for class names, All caps for constants
    c. There are no constants in python. You can only show your intention by writing All Capital letters.
    d.

```
# Constants (ALL CAPS)
PI = 3.14159
MAX_CONNECTIONS = 100

# Function (Snake case)
def calculate_circle_area(radius):
    return PI * radius * radius

def calculate_interest(account_balance, interest_rate):
    return account_balance * interest_rate / 100

# Variables (Snake case)
user_name = "Alice"
account_balance = 5000.0
radius = 10
interest_rate = 5
```

2. **DataTypes (**All these are classes**)**
    a. **Built In data types:**
        i. Numeric – int, float, complex, bool
        ii. Sequence Type – string, list, tuple, Range
            1. There are no characters in Python. They are also strings
        iii. Mapping Type – dictionary (Key and Value pairs)
        iv. Set Type – Set
        v. None
            1.
```
value = None
print(value)
```
    b. **Custom data types:**
        i. **User defined classes:** Custom data types are user-defined classes that can define their own attributes and methods.
3. **Numeric Data types:** Numeric types are used to store numerical values.
    a. **int**: Integer values (whole numbers).
    b. **float**: Floating-point numbers (decimal values).
    c. **complex**: Numbers with real and imaginary parts.
    d. **bool**: Boolean values (True or False).
    e. **Examples:**
        i.
```
# Numeric types
integer_number = 10       # int
floating_number = 3.14    # float
complex_number = 2 + 3j   # complex
is_valid = True           # bool
```
4. Sequence Type: Sequence types represent ordered collections of items.
    a. **string**: A sequence of characters. Strings are immutable in Python.
    b. **list**: An ordered, mutable collection of items.
    c. **tuple**: An ordered, immutable collection of items.

i. Key difference between list and tuple is - **Lists** are mutable and Tuples are immutable.

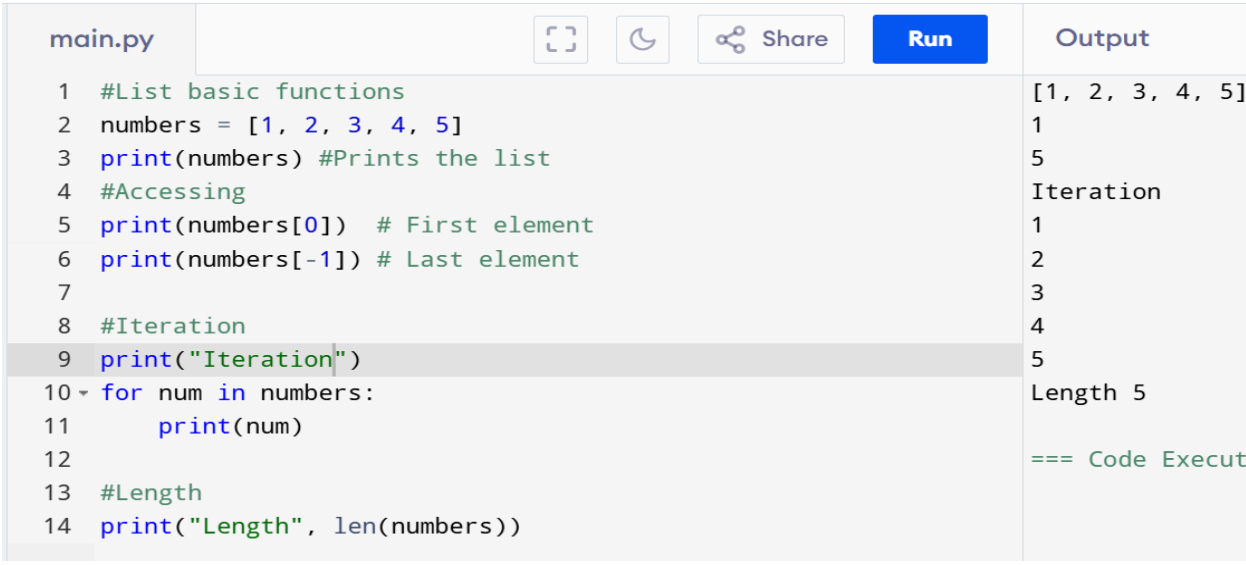
d. **range**: Represents a sequence of numbers, often used in loops.

e.

```
# Sequence types
text = "Hello, World!"          # string
numbers_list = [1, 2, 3, 4]     # list
coordinates = (10, 20, 30)      # tuple
range_of_numbers = range(5)     # range (0, 1, 2, 3, 4)
```

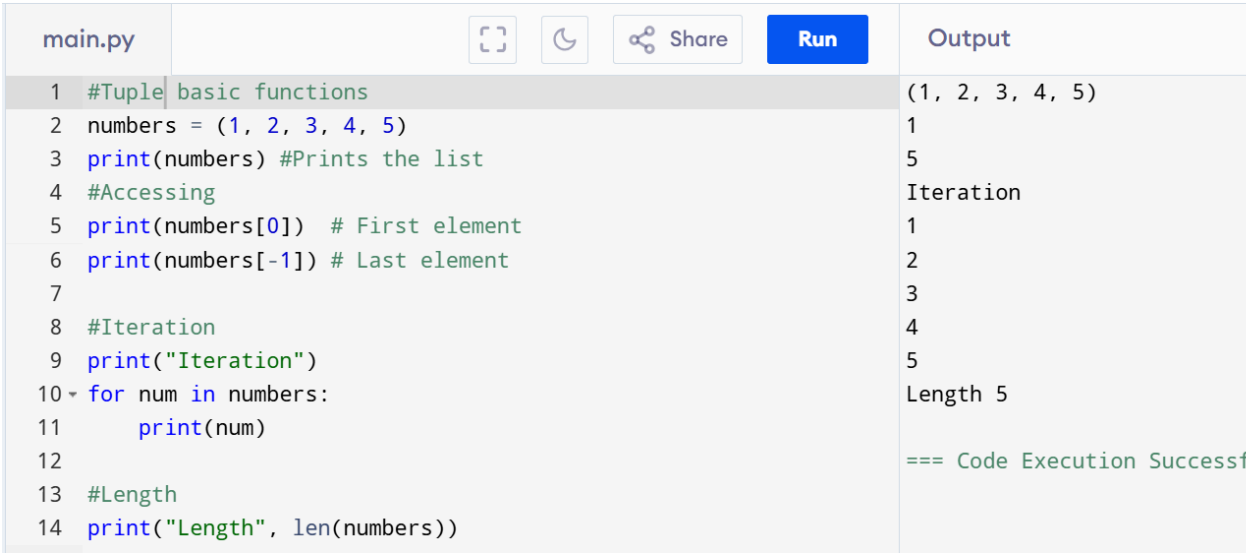
f. List basic functions:

i.



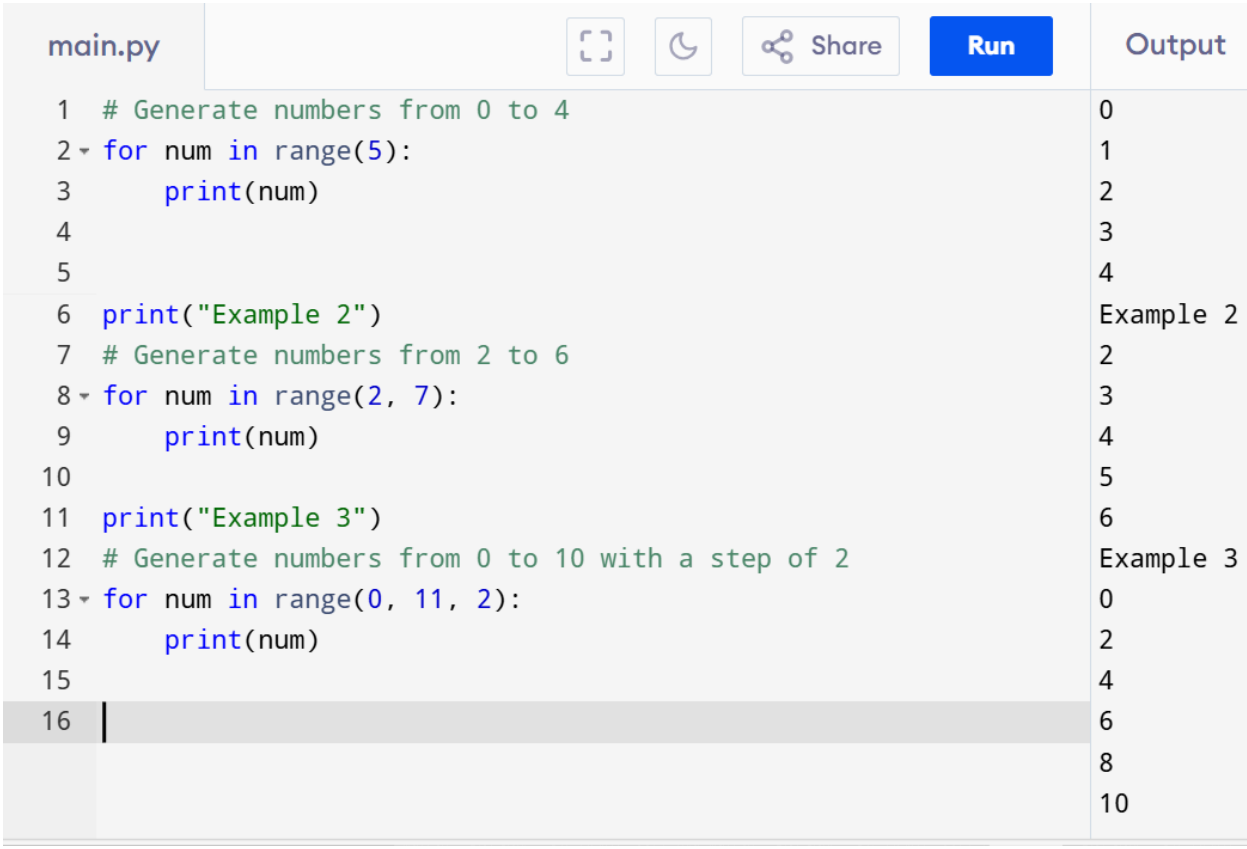
g. Tuple basic functions:

i.



h. Range examples:

i.



ii.

5. **Mapping Type:** Mapping types store key-value pairs.
   a. **Dictionary**: A collection of key-value pairs where keys must be unique.
   b. Basic operations on Dictionary:

c.

```
# Creating a dictionary
person = {"name": "Alice", "age": 25, "city": "New York"}
print("Original dictionary:", person)

# Accessing values using keys
print("Name:", person["name"])
print("Age:", person.get("age"))  # Using get() method

# Adding or updating key-value pairs
person["profession"] = "Engineer"  # Add new key-value
person["age"] = 26  # Update existing value
print("Updated dictionary:", person)

# Iterating through keys and values
person = {"name": "Alice", "age": 25, "city": "New York"}
for key, value in person.items():
    print(f"{key}: {value}")
```

```
Original dictionary: {'name': 'Alice', 'age': 25, 'city': 'New
    York'}
Name: Alice
Age: 25
Updated dictionary: {'name': 'Alice', 'age': 26, 'city': 'New York',
    'profession': 'Engineer'}
name: Alice
age: 25
city: New York

=== Code Execution Successful ===
```

6. **Set Type:** A **set** is an unordered collection of unique items.

a.

```
# Set type
unique_numbers = {1, 2, 3, 4, 5}
```

b. Order of the elements might not be same as the order of elements as in declaration

c. Even if you give duplicates in declaration of set, it will store only one value.

7. **Introduce Type function**

a.

```
x = 5
print(type(x))  # Output: <class 'int'>

y = "Hello"
print(type(y))  # Output: <class 'str'>

z = [1, 2, 3]
print(type(z))  # Output: <class 'list'>
```

8. **Id function**

a.

```
a = 10
b = 10
print(id(a))
print(id(b))
print(id(10))
```

```
136341626880296
136341626880296
136341626880296

=== Code Execution Successful ===
```

Numeric Types:

1. **Conversion from one type to another**

a.

```
# Conversions
x = 10          # int
y = float(x)    # Converts int to float
z = complex(x)  # Converts int to complex

print(y)        # 10.0
print(z)        # (10+0j)
```

2. Similarly conversions can be done for other data types also.

3.

	int	float	complex	bool	list	tuple	dictionary	set	string
int	✓	✓	✓	✓	✗	✗	✗	✗	✓
float	✓	✓	✓	✓	✗	✗	✗	✗	✓
complex	✗	✗	✓	✓	✗	✗	✗	✗	✓
bool	✓	✓	✓	✓	✗	✗	✗	✗	✓
list	✗	✗	✗	✓	✓	✓	✗	✓	✓
tuple	✗	✗	✗	✓	✓	✓	✗	✓	✓
dictionary	✗	✗	✗	✓	✓	✓	✓	✓	✓
set	✗	✗	✗	✓	✓	✓	✗	✓	✓
string	✓*	✓*	✓*	✓	✓	✓	✗	✓	✓

a. ✓: Conversion is supported directly using type casting (e.g., int(x), float(x)).

b. ✓*: Supported if the string is formatted correctly (e.g., int("123"), float("3.14")).

c. ✗: Conversion is not supported directly and may raise a TypeError or ValueError.

d. NOTE: Only homogeneous lists/tuples i.e., lists/tuples containing values of same data types can be converted into sets.

e.

Basic Input and Output

1.
```python
name = input("Enter your name: ")  # Prompt user
print("Hello, " + name)  # Use the input
```
2. Type conversion - int(), float()

Different Number Systems in Python:

1. Decimal, Binary, Octagonal, Hexadecimal system

```
print(bin(93))
print(0b1011)
print(oct(15))
print(hex(19))
print(0x13)
```

```
0b1011101
11
0o17
0x13
19
```

2.

Operators

1. Arithmetic operators - (+, -, *, /, //, ** etc)
    a. / is float division
    b. // is integer division
2. Assignment operators - (=, +=, -=, *= etc)
    a. a,b = 2, 3
    b. n = -n
3. Relational operators - (<, >, <=, >=, ==, !=)
4. Logical operators - (and, or, not)
    a. Truthy and Falsy values in Python # Falsy: 0, 0.0, '', None, False, [], {}, (), set()
    b. Truthy: Non-zero numbers, non-empty strings, non-empty collections, True

```
# Using `and`
print(5 and 10)         # 10 (Both are truthy; returns the second value)
print(0 and 10)         # 0 (First is falsy; returns it immediately)
print('Hello' and 'Hi') # 'Hi' (Both are truthy; returns the second value)
print('' and 'Hi')      # '' (First is falsy; returns it)

# Using `or`
print(5 or 10)          # 5 (First is truthy; returns it immediately)
print(0 or 10)          # 10 (First is falsy; evaluates and returns the second)
print('Hello' or 'Hi')  # 'Hello' (First is truthy; returns it)
print('' or 'Hi')       # 'Hi' (First is falsy; returns the second)
```

    c.

d.

```
# Using `not`
print(not 5)            # False (5 is truthy)
print(not 0)            # True (0 is falsy)
print(not 'Hello')      # False ('Hello' is truthy)
print(not '')           # True (Empty string is falsy)

# Combining `and`, `or`, and `not`
print((5 and 0) or (10 and 15))  # 15 (First part is falsy, evaluates the second)
print(not (5 and 0))             # True (5 and 0 is falsy, so `not` makes it True)
print((0 or 'Hi') and 'Hello')   # 'Hello' (First part evaluates to 'Hi', which is truthy)
print(not ('' or 0))             # True ('' or 0 is falsy, so `not` makes it True)
```

Explanation of Results:

1. **and**: Returns the first falsy value if encountered; otherwise, the last truthy value.
2. **or**: Returns the first truthy value if encountered; otherwise, the last falsy value.
3. **not**: Negates the truth value of the operand.

5. Bitwise operator - Total 6 operators - Complement(~), &, |, ^, << (left shift), >> (right shift)

a.

```
print(~12)                -13
print(12 | 13)            13
print( 12 & 13)           12
print(12 ^ 13)            1
print (23 >> 1)           11
print (51 << 1)           102
```

   i. **Bitwise NOT (~):** Inverts all the bits of a number (flips 0 to 1 and vice versa).
   ii. **Bitwise AND (&)**: Performs an AND operation on each bit of two numbers. The result is 1 if both bits are 1; otherwise, 0.
   iii. **Bitwise OR (|) :** Performs an OR operation on each bit of two numbers. The result is 1 if either bit is 1; otherwise, 0.
   iv. **XOR (^)**: Performs an XOR operation on each bit of two numbers. The result is 1 if the bits are different; otherwise, 0
   v. **Left Shift (<<):** Shifts the bits of a number to the left by the specified number of positions, effectively multiplying the number by 2 for each shift.
   vi. **Right Shift (>>):** Shifts the bits of a number to the right by the specified number of positions, effectively dividing the number by 2 for each shift.

6. Membership operators: in, not in

9. Basic Troubleshooting (Syntax errors, Indentation)

a. Syntax error: Common Causes

i. Missing colons (:) in if, for, while, etc.

ii. Using invalid characters. ?

iii. Incorrect use of parentheses, brackets, or quotes.

b. **Indentation error:** Generally 4 spaces i.e one tab

c. **Type errors:**

i. Example adding a number to a string

d.

```
# Example with multiple issues
if 10 > 5  # SyntaxError: Missing ':'
print("This is a syntax error")  # IndentationError: expected an indented block
result = 5 + "text"  # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Class 3 and Class 4:

A control statement is a programming construct that dictates the flow of execution in a program. It determines the order in which statements are executed based on certain conditions or repetitions. By using control statements, **you can make decisions, repeat actions, or jump to specific parts** of your program.

Categories of Control Statements:

Control statements are broadly classified into three categories:

1. Conditional Statements
   Used to execute specific blocks of code based on a condition.
   - Example: if, else, elif (or else if), switch (in some languages).
2. Looping Statements
   Allow a block of code to be executed repeatedly, either for a fixed number of times or until a condition is met.
   - Example: for, while, do-while.
3. Jump Statements
   Control the flow of loops or exit a block of code prematurely.
   - Example: break, continue, return, pass.

Purpose of Control Statements

1. Decision-making: Decide which path of code to execute (e.g., if-else).
2. Repetition: Repeat tasks (e.g., using loops).
3. Branching: Exit or skip specific parts of code when needed (break, continue).

1. Conditional Statements

Conditional statements allow decision-making in your code, where specific blocks of code execute based on conditions.

1. **If, else and elif:**
    a.
```
if condition1:
    # Code if condition1 is true
elif condition2:
    # Code if condition2 is true
else:
    # Code if none of the above conditions are true
```
1. **Loops: For and While**
    a. Loops are used to execute a block of code repeatedly until a specific condition is met.
    b. For Loop:
        i.
```
for item in sequence:
    # Code to execute for each item
```
        ii.
```
for i in range(5):
    print(i)  # Prints 0 to 4
```
    c. While:
        i.
```
count = 0
while count < 5:
    print(count)
    count += 1
```

Break, continue, pass

1. **Break: Exits the loop prematurely when a specific condition is met.**

```
for i in range(10):
    if i == 5:
        break  # Stops the loop when i equals 5
    print(i)
```

a.

2. **Continue: Skips the rest of the current loop iteration and moves to the next iteration.**

```
for i in range(5):
    if i == 3:
        continue  # Skips the iteration when i equals 3
    print(i)
```

a.

3. **Pass:** A placeholder that does nothing and allows for syntactically correct empty code blocks.

```
for i in range(5):
    if i == 3:
        pass  # Placeholder
    print(i)
```

a.

Nested loops and Nested conditionals:

1. **Nested Loops:** A **nested loop** is a loop inside another loop. The inner loop runs completely for every single iteration of the outer loop.

```
# Nested loops for multiplication table
for i in range(1, 4):  # Outer loop
    for j in range(1, 4):  # Inner loop
        print(f"{i} x {j} = {i * j}")
```

a.

2. Nested Conditions:

a.

```
# Nested conditional statements
age = 20
has_id = True

if age >= 18:
    if has_id:
        print("You are allowed entry.")
    else:
        print("Please provide an ID.")
else:
    print("You must be at least 18 years old.")
```

3. Nested Loops with Conditions:

a.

```
# Nested loops with conditionals
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]

for row in matrix:  # Outer loop iterating through rows
    for number in row:  # Inner loop iterating through numbers in each row
        if number % 2 == 0:  # Conditional to check for even numbers
            print(f"Even number found: {number}")
```

b. Example: Finding Prime Numbers in a given range:

i.

```
# Find prime numbers between 2 and 20
for num in range(2, 21):  # Outer loop to iterate through numbers
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):  # Inner loop for divisors
        if num % i == 0:  # Check if divisible
            is_prime = False
            break
    if is_prime:  # Conditional to check if the number is prime
        print(f"{num} is a prime number")
```

Functions:

1. **Definition**: What are functions and why are they useful?
    a. A **function** is a reusable block of code designed to perform a specific task. Functions take inputs (called parameters or arguments), process them, and return an output or perform an action. Functions are a fundamental concept in programming and are used to encapsulate logic, making the code modular, readable, and reusable.
    b. Why are they useful?
        i. Reusability of code - Can use the same code multiple times.
        ii. Modularity: By breaking your code into smaller, logical units (functions), it becomes easier to understand, debug, and maintain.
        iii. Readability
        iv. Easier debugging, scalability and testing
2. **Built-in vs. User-defined functions**

Feature	Built-in Functions	User-defined Functions
Definition	Predefined by the programming language.	Created by the programmer.
Examples	`print()`, `len()`, `max()`, `abs()`	`def calculate_area(length, width):`
Purpose	General-purpose, widely used tasks.	Specific to the programmer's needs.
Code Requirement	No coding required; directly usable.	Must be written explicitly by the user.

    a.
    b. **User defined function example:**
        i.

```
def calculator(a, b):
    print(a + b)
    print(a - b)
    print(a * b)

calculator(2,3)
```

```
5
-1
6

=== Code Execution Successful ===
```

        ii. After defining a function, you have to call it to use that function as shown in the above example.
3. **Components of a function**: Name of a function, parameters, body
4. **Difference between arguments and parameters:**
    a. **Parameters**: Parameters are the placeholders or variables **declared in the function definition**. They define what inputs the function can accept. Parameters act as variables that will receive the values passed to the function when it is called
        i.

```
def greet(name):  # 'name' is the parameter
    print(f"Hello, {name}!")
```

        ii. name is a parameter here
    b. **Arguments:** Arguments are the **actual values passed to the function** when it is called. These values are assigned to the corresponding parameters.

i.
```
greet("Alice")  # "Alice" is the argument
```

5. **Arguments Types**
   a. Positional arguments: These are arguments passed to a function based on their position in the function definition.

   i.
```
def add(a, b):
    return a + b

result = add(5, 3)  # 5 is assigned to 'a', 3 to 'b'
print(result)  # Output: 8
```

   b. Keyword arguments: These are arguments passed by explicitly specifying their parameter names, making the order irrelevant.

   i.
```
def introduce(name, age):
    print(f"My name is {name} and I am {age} years old.")

introduce(age=25, name="Bob")  # Order doesn't matter
```

   c. Default arguments: These are arguments that have default values. If no value is provided for the argument during the function call, the default value is used.

   i.
```
def greet(name="Guest"):
    print("Hello", name)

greet()  # Uses default value: "Guest"
greet("Alice")  # Overrides default: "Alice"
```
```
Hello Guest
Hello Alice

=== Code Execution Successful ===
```

   d. Arbitrary arguments: Allows a function to accept any number of positional arguments. The arguments are packed into a tuple.

```
def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result

print(multiply(2, 3, 4))  # Output: 24
print(multiply(5, 6))     # Output: 30
```

i.

1. This code returns multiplication of all the arguments passed.

e. Keyword arbitrary arguments

```
def total_sum (a, b, **kargs):
    sum = a + b
    for i, j in kargs.items():
        print(i, j) # Here i, j are key and value pairs
        sum += j
    return sum

sum = total_sum(5, 6, num1 = 23, num2 = 45, num3 = 22)
print("This is the final sum", sum)
```

```
num1 23
num2 45
num3 22
This is the final sum 101

=== Code Execution Successful ===
```

i.

6. **Scope and Lifetime**: **Scope** refers to the region of a program where a variable is accessible, while **lifetime** refers to how long the variable exists in memory.
   a. **Local vs. Global variables**
      i. **Global Scope**: Variables declared outside any function are global and accessible throughout the program.

```
global_var = 20  # Global variable

def my_function():
    print(global_var)  # Accessing global variable

my_function()
```

1.

      ii. **Local Variables:** Variables declared inside a function have a local scope and are accessible only within that function.

1.
```
def my_function():
    local_var = 10  # Local variable
    print(local_var)
my_function()
# print(local_var)  # Error: local_var is not defined outside the function
```
2. Another Example of Local variable
3.
```
num1 = 10

def func():
    num1 = 25
    print("Num1 value in function", num1)

func()
print(num1) #global num1 value is not effected by change of
    value inside function. Because num1 in function is a new
    local variable which is not same as num1 in global scope
```
```
Num1 value in function 25
10

=== Code Execution Successful
```

b. **Lifetime**
   i. Local variables exist only as long as the function executes.
   ii. Global variables persist throughout the program's execution.

c. **Using global keyword:** Used to modify a global variable inside a function

i.
```
counter = 0

def increment():
    global counter
    counter += 1

increment()
print(counter)  # Output: 1
```

d. Using global and local variables in a single function using globals function.
   i. The globals() function retrieves a dictionary of all global variables, allowing explicit control over global scope.

ii.

```
num1 = 10  # Global variable

def use_globals_and_locals():
    num1 = 5  # Local variable
    print("Local num1:", num1)  # Prints the local
        variable

    # Access and modify the global num1 using globals
        () function
    globals()['num1'] += 1
    print("Global num1 after modification:", globals
        ()['num1'])

use_globals_and_locals()
print("Global num1 outside the function:", num1)
```

```
Local num1: 5
Global num1 after modification: 11
Global num1 outside the function: 11

=== Code Execution Successful ===
```

7. **Types of functions:**
    a. **Void functions:** Void functions perform an action but do not return any value. They typically return None by default unless explicitly specified otherwise.
        i. Does not include a return statement (or has a return without a value).
        ii.

```
def greet(name):
    print(f"Hello, {name}!")  # Performs an action but doesn't return anything

result = greet("Alice")  # Output: Hello, Alice!
print(result)  # Output: None (default return value)
```

    b. **Functions with return values**: Functions with return values process some data and return a result using the return keyword. The returned value can be used or stored in the calling code.
        i. A function can return any data type: integers, strings, lists, objects, or even other functions.
            1.

```
def add(a, b):
    return a + b  # Returns the sum of a and b

result = add(5, 3)
print(result)  # Output: 8
```

            2. **Returning Multiple Values**:
                a.

```
def calculate(a, b):
    return a + b, a - b  # Returns a tuple with the results

sum_result, diff_result = calculate(10, 5)
print(sum_result, diff_result)  # Output: 15 5
```

c. **Recursive functions:** A recursive function is a function that calls itself in its definition. It must include a base condition to avoid infinite recursion.
   i. **Use Cases**: Solving problems that can be broken into smaller subproblems (e.g., factorials, Fibonacci sequence, traversals).
   ii. Example: Recursive function for Factorial

```python
def factorial(n):
    if n == 0 or n == 1:  # Base case
        return 1
    return n * factorial(n - 1)  # Recursive call

print(factorial(5))  # Output: 120
```

   iii.
   iv. Example: Fibonacci of a number

```python
def fibonacci(n):
    if n <= 1:  # Base case
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive call

print(fibonacci(6))  # Output: 8
```

   v.
   vi. Print 1 to n numbers with recursion - WITHOUT using loops

```python
def print_nums(n):
    if n == 0:
        return
    print(n)
    print_nums(n-1)

print_nums(10)
```

```
10
9
8
7
6
5
4
3
2
1
```

   vii.

8. **Lambda functions**: A **lambda function** in Python is a small, anonymous function that is defined using the lambda keyword. It can have any number of arguments but only one expression. Lambda functions are typically used for short, simple operations and are often used in combination with functions like map(), filter(), and sorted().
   a. **Key Characteristics**
      i. **Anonymous**: Lambda functions do not have a name.
      ii. **Single Expression**: They can only contain a single expression, not multiple statements.

iii. **Short and Concise**: Used for small tasks, often in one-liners.
iv. **Usage Scope**: Typically used in higher-order functions or short-term use cases.

b. Syntax: **lambda arguments: expression**

i.
```
# Regular function
def square(x):
    return x * x

# Equivalent lambda function
square = lambda x: x * x

print(square(5))  # Output: 25
```

c. Lambda function with multiple arguments

i.
```
add = lambda a, b: a + b
print(add(3, 7))  # Output: 10
```

d. Lambda with no arguments:

i.
```
say_hello = lambda: "Hello, World!"
print(say_hello())  # Output: Hello, World!
```

e. Example to sort a list of tuples based on second element

i.
```
# Sort a list of tuples by the second element
data = [(1, 'b'), (2, 'a'), (3, 'c')]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)  # Output: [(2, 'a'), (1, 'b'), (3, 'c')]
```

ii. Where 'sorted' is an inbuilt function which can be used to sort a list

9. **Higher order functions - map(), filter(), reduce()**

a. map() - The map() function is used to **apply a given function to each item in an iterable (e.g., list, tuple)** and returns a map object (an iterator). It does not modify the original iterable but generates a transformed version.

i. Syntax:

```
map(function, iterable)
```

ii. **Key Points**:

1. The first argument is a function (can be a lambda function or a normal function).
2. The second argument is an iterable (like a list, tuple, etc.).
3. The result can be converted to a list or another iterable.

iii. **Example**: Squaring every element in the numbers list using a map.

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

1.
2. The result is converted to a list for display.

b. Filter()

i. The filter() function is used to **filter items from an iterable based on a condition** provided by a function. It returns a filter object (an iterator) containing only the elements that satisfy the condition.'

ii. **Key Points**:

1. The function must return True or False for each element.
2. Only elements for which the function returns True are included in the result.

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4, 6]
```

iii.

c. reduce()

i. The reduce() function is used to **apply a rolling computation to sequential pairs of items** in an iterable, reducing it to a single cumulative value. It is part of the functools module and must be explicitly imported.

ii. Syntax:

```
from functools import reduce
reduce(function, iterable[, initializer])
```

1.

iii. **Key Points**:

1. The function must take two arguments: the accumulated result and the next item.

2. It applies the function cumulatively to the items in the iterable.
3. Optionally, an initializer can be provided to start the reduction.

10. **Decorators**: A **decorator** in Python is a design pattern that allows you to **modify or extend the behavior of a function** without changing its actual implementation. Decorators are often used to add functionality like logging, authentication, or performance monitoring in a reusable and clean way.
    a. A decorator is a function that **takes another function as an argument**, enhances or modifies it, and returns a new function.
    b. The @decorator_name syntax is used to apply a decorator to a function.
    c. How does a decorator work?
        i. A decorator is a higher-order function (a function that takes another function as input).
        ii. It wraps the input function in another function (called the wrapper).
        iii. The wrapper adds functionality before or after calling the original function.
    d. Example:
        i.

```
def decorator_function(func):
    def wrapper():
        print("Lines before function call")
        func()
        print("Lines after function call completion")
    return wrapper


@decorator_function  #Taging decorator to printer function
def printer():
    print("Printing in progress.....")

printer() #Function
```

```
Lines before function call
Printing in progress.....
Lines after function call completion

=== Code Execution Successful ===
```

11. Best Practices for writing functions
    a. Single Responsibility Principle: A function should perform **only one task or responsibility**. This makes functions easier to understand, test, and maintain.
    b. Writing reusable and modular code: Functions should be designed to be **reused** in different parts of the program and **modular** enough to be independent of specific contexts.
    c. Proper naming conventions: Function names should clearly convey what the function does, making the code self-documenting.
    d. Keeping functions small and focused: A function should do one thing and do it well. Keeping functions small makes them easier to read, test, and debug.

Basic Inbuilt Functions in Python:

String inbuilt functions: Strings in Python are Immutable.

Accessing string elements - Strings are indexed, starting from 0 for the first character. Negative indices start from -1 for the last character.	`s = "Python"`
	`print(s[0])  # Output: P`
	`print(s[-1]) # Output: n`

Slicing Strings	`s = "Python"`
	`print(s[0:3])  # Output: Pyt (0 to 2)`
	`print(s[:3])  # Output: Pyt (start is 0 by default)`
	`print(s[3:])  # Output: hon (end is the length by default)`
	`print(s[-3:])  # Output: hon (last 3 characters)`

Concatenation: Adding 2 strings	`str1 = "Hello"`
	`str2 = "World"`
	`result = str1 + " " + str2`
	`print(result)  # Output: Hello World`

Repetition: Can multiply with an integer	`str1 = "Hi! "` `print(str1 * 3)  # Output: Hi! Hi! Hi!`
strip() - Removes whitespace (or specified characters) from both ends of a string.	`text = "  Hello, World!  "` `print(text.strip())  # Output: "Hello, World!"`
lower() - Converts all characters in a string to lowercase	`text = "Hello"` `print(text.lower())  # Output: "hello"`
upper() - Converts all characters in a string to uppercase	`text = "Hello"` `print(text.upper())  # Output: "HELLO"`
find(substring) - Returns the index of the first occurrence of a substring; returns -1 if not found.	`text = "Hello, World!"` `print(text.find("World"))  # Output: 7`

replace(old, new) - Replaces all occurrences of a substring with another substring.	`text = "I love Python"` `print(text.replace("Python", "coding"))  # Output: "I love coding`
split(delimiter) - Splits a string into a list based on a delimiter	`text = "a,b,c"` `print(text.split(","))  # Output: ['a', 'b', 'c']`
",".join(iterable) - Joins elements of an iterable (e.g., list) into a single string, separated by the specified delimiter.	`items = ["a", "b", "c"]` `print(",".join(items))  # Output: "a,b,c"`
startswith(prefix) - Returns True if the string starts with the specified prefix; otherwise, False.	`text = "Hello"` `print(text.startswith("He"))  # Output: True`
endswith(suffix) - Returns True if the string ends with the specified suffix; otherwise, False.	`text = "Hello"` `print(text.endswith("lo"))  # Output: True`
count(substring) - Counts occurrences of a substring in a string	`text = "banana"` `print(text.count("a"))  # Output: 3`

List Methods:

Accessing and Slicing	`numbers = [1, 2, 3, 4, 5]`
	`print(numbers[1])  # Output: 2`
	`print(numbers[1:4])  # Output: [2, 3, 4]`

len() - Returns the number of elements in a list	`numbers = [1, 2, 3]` `print(len(numbers))  # Output: 3`

in - Checks if an element exists in the list.	`numbers = [1, 2, 3]` `print(2 in numbers)  # Output: True`

Concatenation - Combines two lists.	`list1 = [1, 2]`
	`list2 = [3, 4]`
	`print(list1 + list2)  # Output: [1, 2, 3, 4]`

append(item) - Adds an item to the end of the list.	`numbers = [1, 2]`
	`numbers.append(3)`
	`print(numbers)  # Output: [1, 2, 3]`

extend(iterable) - Extends the list by appending elements from another iterable.	`numbers = [1, 2]`
	`numbers.extend([3, 4])`
	`print(numbers)  # Output: [1, 2, 3, 4]`

insert(index, item) - Inserts an item at a specified position.	`numbers = [1, 3]`
	`numbers.insert(1, 2)`
	`print(numbers)  # Output: [1, 2, 3]`

remove(element) - Removes the first occurrence of an element.	`numbers = [1, 2, 3]`
	`numbers.remove(2)`
	`print(numbers)  # Output: [1, 3]`

pop(index) - Removes and returns the element at the specified index. Default: last item.	`numbers = [1, 2, 3]`
	`print(numbers.pop(1))  # Output: 2`
	`print(numbers)  # Output: [1, 3]`

index(element) - Returns the index of the first occurrence of an element.	`numbers = [1, 2, 3]` `print(numbers.index(2))  # Output: 1`

reverse() - Reverses the list in place.	`numbers = [1, 2, 3]`
	`numbers.reverse()`
	`print(numbers)  # Output: [3, 2, 1]`

sort(reverse=False) - Sorts the list in ascending order (default) or descending order (if reverse=True).	`numbers = [3, 1, 2]`
	`numbers.sort()`
	`print(numbers)  # Output: [1, 2, 3]`

clear() - Removes all elements from the list.	`numbers = [1, 2, 3]`
	`numbers.clear()`
	`print(numbers)  # Output: []`

Set Methods:

add(element) - Adds an element to the set.	`s = {1, 2}`
	`s.add(3)`
	`print(s)  # Output: {1, 2, 3}`

remove(element) - Removes the specified element. Raises KeyError if not found	`s = {1, 2, 3}`
	`s.remove(2)`
	`print(s)  # Output: {1, 3}`

discard(element) - Removes the specified element but does not raise an error if not found.	`s = {1, 2, 3}`
	`s.discard(4)`
	`print(s)  # Output: {1, 2, 3}`

pop() - Removes and returns an arbitrary element.	`s = {1, 2, 3}`
	`print(s.pop())  # Output: 1 (or another element)`
	`print(s)  # Output: {2, 3}`

clear() - Removes all elements from the set.	`s = {1, 2, 3}`
	`s.clear()`
	`print(s)  # Output: set()`

union(other_set) - Returns a set containing all elements from both sets.	`s1 = {1, 2}`
	`s2 = {2, 3}`
	`print(s1.union(s2))  # Output: {1, 2, 3}`

intersection(other_set) - Returns elements common to both sets.	`s1 = {1, 2}`
	`s2 = {2, 3}`
	`print(s1.intersection(s2))  # Output: {2}`

difference(other_set) - Returns elements in the first set but not in the second.	`s1 = {1, 2}`
	`s2 = {2, 3}`
	`print(s1.difference(s2))  # Output: {1}`

symmetric_difference(other_set) - Returns elements in either set, but not in both.	`s1 = {1, 2}`
	`s2 = {2, 3}`
	`print(s1.symmetric_difference(s2))  # Output: {1, 3}`

issubset(other_set) - Checks if all elements of the set are in another set.	`s1 = {1, 2}`
	`s2 = {1, 2, 3}`
	`print(s1.issubset(s2))  # Output: True`

issuperset(other_set) - Checks if the set contains all elements of another set.	`s1 = {1, 2, 3}`
	`s2 = {1, 2}`
	`print(s1.issuperset(s2))  # Output: True`

isdisjoint(other_set) - Returns True if the sets have no elements in common.	`s1 = {1, 2}`
	`s2 = {3, 4}`
	`print(s1.isdisjoint(s2))  # Output: True`

#Ternary operators say that you will discuss in if-else statements.

```
a, b = 10, 20
print("Both a and b are equal" if a == b else "a
    is greater than b" if a > b else "b is
    greater than a")
```

```
b is greater than a

=== Code Execution Successful ===
```

1. Programme to check largest number in python

#Other Points: Free and Open source, Dynamically typed. Front-end and backend development.

#Update python command

#Data Type definition and each one size.

#Functions definition, ternary operator.

#Mentor | 10000Coders

#Dictionary

#For logical operators - And, Or and Not. Draw truth tables.

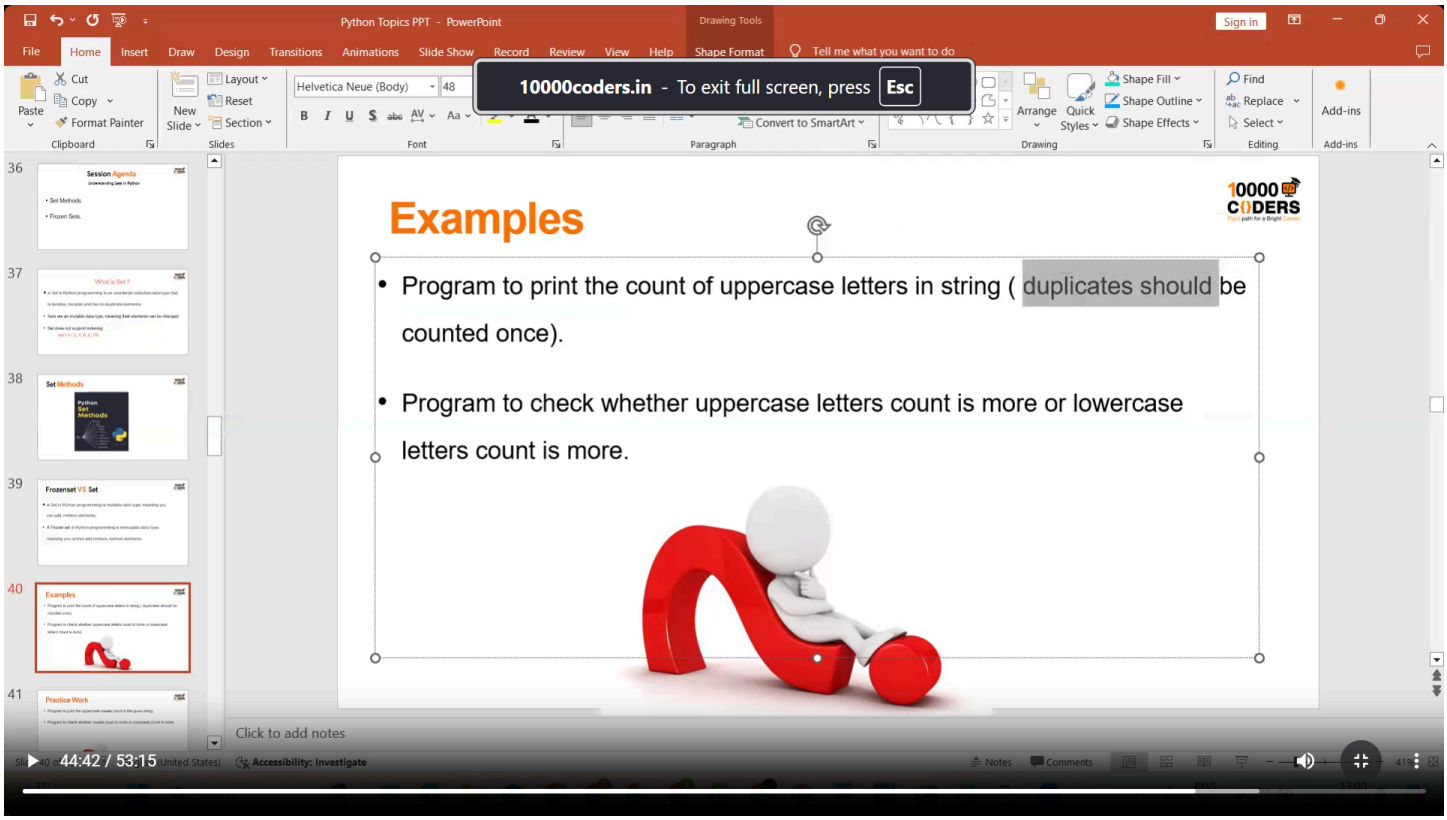
#iloc and loc differences.

Variables topic: Definition, Initialization, reassignment, user input, type

int() - Type casting function.

Frozen set.

Sets questions:



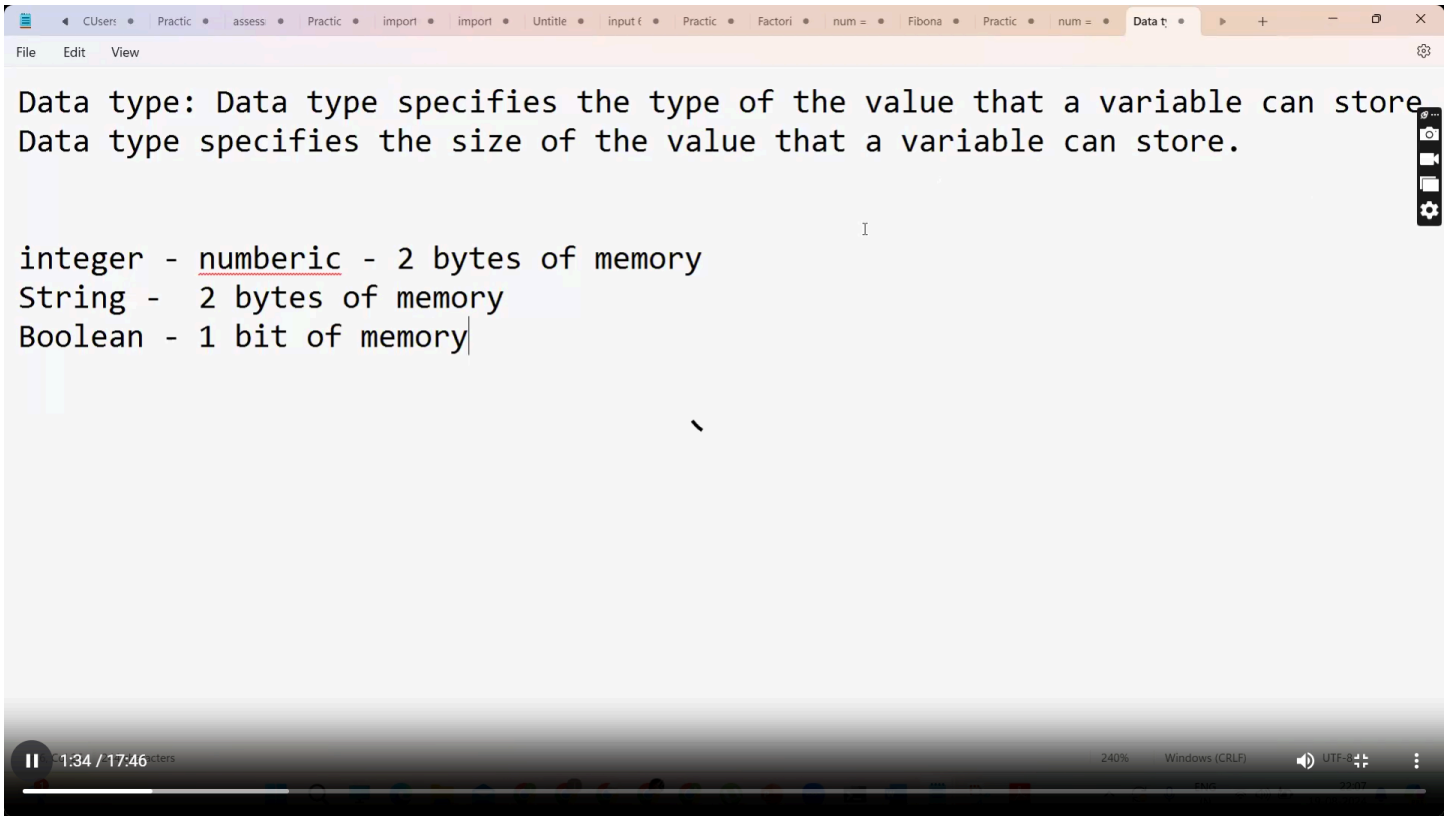
We can use logical operators to check if more than one condition is true.

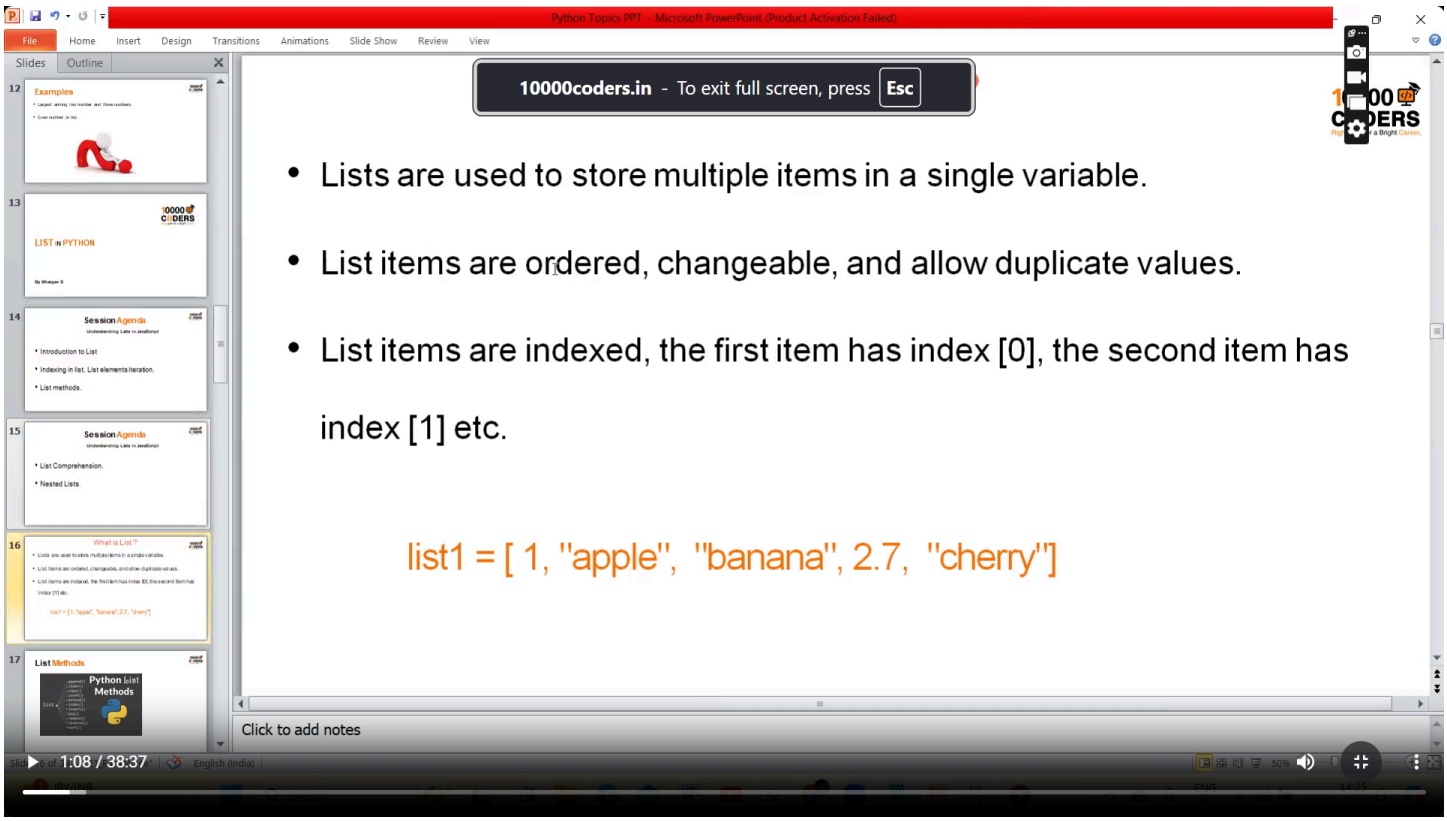
Bitwise not value for a is -(a+1)

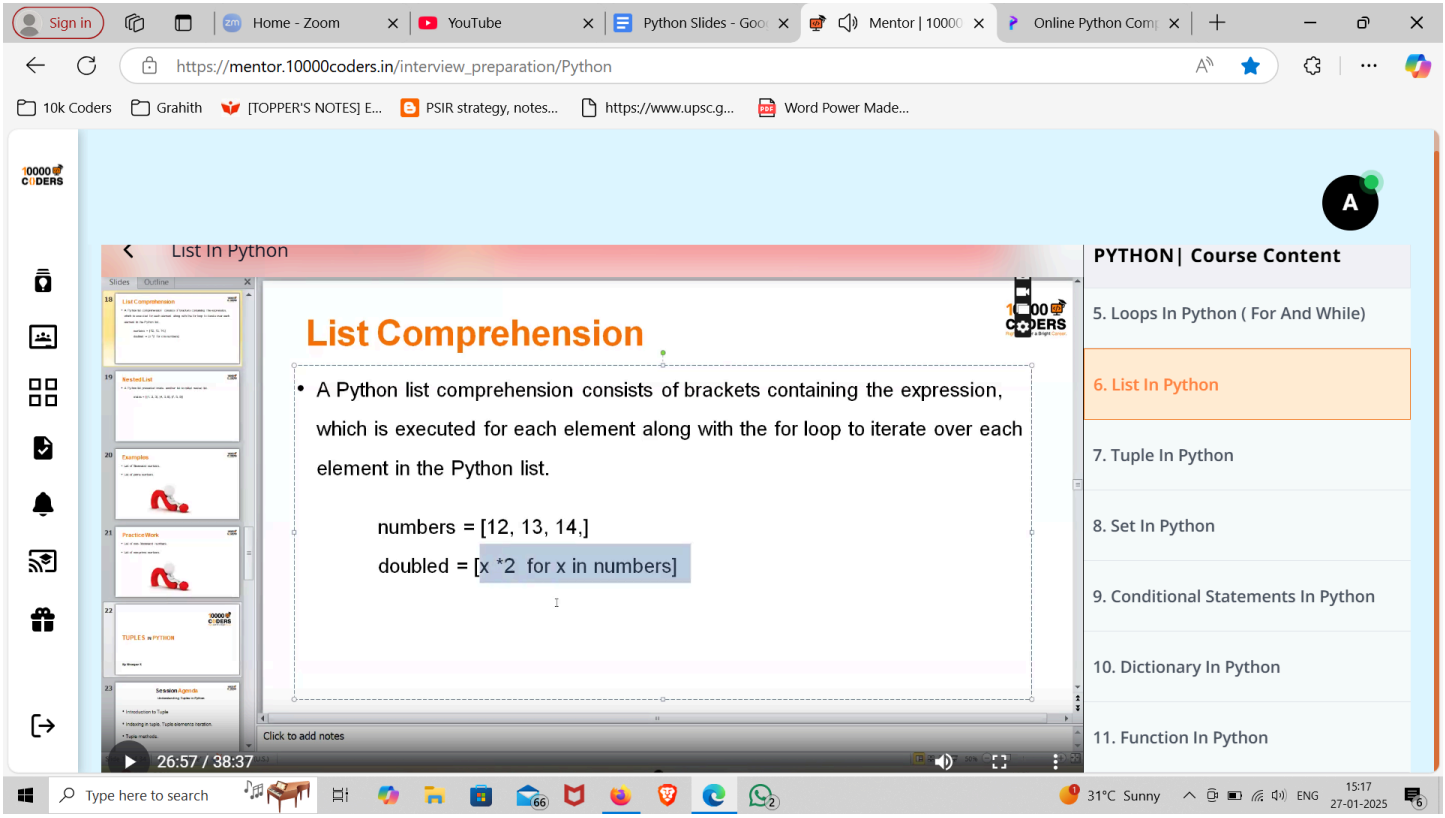
#Inbuilt methods check gfg.

1. List - copy
2. If only giving the first occurrence then how to get all occurrences in inbuilt methods.

1. Write a function which takes a list of numbers and returns sum of prime numbers in it.


File Edit View
Data type: Data type specifies the type of the value that a variable can store.
Data type specifies the size of the value that a variable can store.
integer - numberic - 2 bytes of memory
String - 2 bytes of memory
Boolean - 1 bit of memory
1:34 / 17:46


Python Topics PPT - Microsoft PowerPoint (Product Activation Failed)
10000coders.in - To exit full screen, press Esc
Lists are used to store multiple items in a single variable.
List items are ordered, changeable, and allow duplicate values.
List items are indexed, the first item has index [0], the second item has index [1] etc.
list1 = [1, "apple", "banana", 2.7, "cherry"]
Click to add notes
1:08 / 38:37



Nested lists. Fibonacci numbers generate and store into list. List of prime numbers also.

1. To find duplicated in a list - you can use count inbuilt function

Tuple methods - count, index, len, max, min, sum

Set is iterable.

Frozensets -

Dictionary - Dict keys should be immutable, such as tuples, strings, integers etc. We cannot use mutable (changeable) objects such as lists as keys.

Ages = dict() # You can also create a dictionary like this.

Dictionary: To remove del keyword. Pop, clear

1. For i in dict_name - gives you key values.

2. copy() - It will give a shallow copy of dictionary.
3. dict_name.get() - None if the key is not there.
4. d.items()
5. d.keys(), d.values()
6. update()
7. pop()
8. popitem() - Random
9. Fromkeys

Programme to merge to dictionaries d1 and d2.

```
country_capitals1 = {
   "Germany": "Berlin",
   "Canada": "Ottawa",
   "England": "London"
}
country_capitals2 = {
   "Albania": "Tirana",
   "Australia": "Canberra",
   "Afghanistan": "Kabul"
}
for i in country_capitals2:
    country_capitals1[i] = country_capitals2[i]
print(country_capitals1)
```

Practise questions in dictionary:

1. Programme to print the capitals of the cities in ascending order