Python Basics

1. **help ( )** … help section is where you can find all python functions, symbols ETC, you also find directly specific topic if you already knew your topic example let us find list… >>>help (‘lists’).
2. **Running:** Before you running your code save it with suing python extension (**.PY**) example: **test.py**
3. **Variable**
   - A variable is like a container that stores a value. You give the container a name, and then assign a value to it using the **=** sign.
     - Examples:
       - name = "Alice"
       - age = 25
       - height = 5.6
       - is_student = True
   - **Rules for variable names:**
     - Must start with a letter or underscore (_)
     - Can contain letters, digits, and underscores
     - Case-sensitive (age and Age are different)
4. **Python Data Types**
   - Integer (**int**): Used to store whole numbers (no decimals).

     -
       ```
       apples = 10
       print(apples)      # Output: 10
       print(type(apples)) # Output: <class 'int'>
       ```
   - Float (**float**): Used for decimal or fractional numbers.

     -
       ```
       price = 19.99
       print(price)       # Output: 19.99
       print(type(price))  # Output: <class 'float'>
       ```
   - String (**str**): A sequence of characters, usually inside quotes.

     -
       ```
       message = "Hello, world!"
       print(message)       # Output: Hello, world!
       print(type(message))  # Output: <class 'str'>
       ```
   - Boolean (**bool**): Represents either True or False.

     -
       ```
       is_sunny = True
       print(is_sunny)      # Output: True
       print(type(is_sunny)) # Output: <class 'bool'>
       ```
   - **List:** An ordered collection that can be changed (mutable). Use square brackets.

     -
       ```
       fruits = ["apple", "banana", "cherry"]
       print(fruits[1])      # Output: banana
       print(type(fruits))   # Output: <class 'list'>
       ```

- **Tuple:** Like a list, but **cannot** be changed (immutable). Use parentheses.
  - coordinates = (10.5, 20.3)
    print(coordinates[0])   # Output: 10.5
    print(type(coordinates))# Output: <class 'tuple'>
- Dictionary (**dict**): Stores data as key–value pairs. Use curly braces.
  - person = {"name": "Alice", "age": 25}
    print(person["name"])   # Output: Alice
    print(type(person))   # Output: <class 'dict'>
- **Set:** An unordered collection with no duplicates. Use curly braces.
  - unique_numbers = {1, 2, 3, 3}
    print(unique_numbers)   # Output: {1, 2, 3}
    print(type(unique_numbers)) # <class 'set'>

5. **comments:** In Python, comments are used to explain code and make it more readable. They are ignored by the Python interpreter.
   - **Single-line Comments:** Use the # symbol. Anything after # on that line is a comment.
     - # This is a single-line comment
     - print("Hello, World!") # This prints a message
   - **Multi-line Comments**: There is no official multi-line comment syntax in Python, but you can use multiple # lines.
     - # This is a comment
     - # that spans multiple
     - # lines
     - Or use triple quotes (although this technically creates a string, not a comment — it's useful for docstrings or placeholder comments):

```
"""
This is a multi-line
string that can be used
as a comment (not recommended for actual commenting)
```

6. **Arithmetic Operators:** Arithmetic operators are used to perform mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	7 * 6	42
/	Division	8 / 2	4.0
//	Floor Division	9 // 2	4
%	Modulus (remainder)	9 % 2	1
**	Exponentiation	2 ** 3	8

- Examples:

  -
    ```
    a = 10
    b = 3

    print(a + b)   # 13
    print(a - b)   # 7
    print(a * b)   # 30
    print(a / b)   # 3.333...
    print(a // b)  # 3
    print(a % b)   # 1
    print(a ** b)  # 1000
    ```

7. **Logical Operators:** Logical operators are used to combine conditional statements (typically boolean expressions).

Operator	Description	Example	Result
And	Returns True if both are True	True and False	False
Or	Returns True if one is True	True or False	True
Not	Reverses the boolean value	not True	False

- Examples:
  - And
    - # Both must be True

    -
      ```
      print(True and True)     # True
      print(True and False)    # False

      a = 10
      b = 5
      result = (a > 0) and (b < 10)
      print(result)            # True
      ```
  - or
    - # One must be True

    -
      ```
      print(False or True)     # True
      print(False or False)    # False

      x = 7
      y = 12
      print((x < 5) or (y > 10))  # True
      ```
  - not

    -
      ```
      print(not True)          # False
      print(not False)         # True

      z = 8
      print(not (z == 8))      # False
      ```

8. **String manipulation:** String manipulation in Python involves creating, modifying, and analyzing strings. Here are the **most important concepts and examples** to get you started:
    - Creating Strings: **s = "Hello, World!"**
    - Common String Methods
        - **s.lower()** - Convert to lowercase
          'HELLO'.lower() → 'hello'
        - **s.upper()** - Convert to uppercase
          'hello'.upper() → 'HELLO'
        - **s.strip()** - Remove leading/trailing whitespace
          ' hi '.strip() → 'hi'
        - **s.replace(a, b)** - Replace a with b
          'hi ho'.replace('ho','hello')
        - **s.split(',')** - Split into list by delimiter
          'a,b,c'.split(',') → ['a','b','c']
        - **'--'.join(list)** - Join list into string
          '--'.join(['a','b']) → 'a--b'
        - **s.find('text')** - Find first index of substring
          'hello'.find('l') → 2
    - **Indexing and Slicing**
      ```
      s = "CyberSec"
      print(s[0])    # 'C'
      print(s[-1])   # 'c'
      print(s[1:4])  # 'ybe'
      print(s[:])    # Full copy: 'CyberSec'
      ```
    - **String Formatting**

        -
          ```
          name = "Alice"
          age = 30

          print(f"My name is {name} and I'm {age}")  # f-strings
          print("My name is {} and I'm {}".format(name, age))
          ```
    - **Checking Content**

        -
          ```
          s = "admin123"
          s.isdigit()    # False
          s.isalpha()    # False
          s.isalnum()    # True
          s.startswith("adm")  # True
          s.endswith("123")    # True
          ```

9. **Type conversion:** Type conversion in Python refers to changing the data type of a value from one type to another. There are two main types of conversions:
   - **Implicit Type Conversion:** Python automatically converts one data type to another without user involvement.

     -
       ```
       x = 10     # int
       y = 3.5    # float
       z = x + y  # int + float -> float
       print(z)   # Output: 13.5
       print(type(z))  # <class 'float'> // Python converts int to float during addition to avoid loss of information.
       ```
   - **Explicit Type Conversion (Type Casting):** You manually convert the data type using built-in functions like int(), float(), str(), etc.
     - **int()** – Converts to integer
     - **float()** – Converts to float
     - **str()** – Converts to string
     - **bool()** – Converts to Boolean
     - **list(), tuple(), set()** – Convert to respective collections
     - **Examples**

       -
         ```
         # String to int
         a = "123"
         b = int(a)
         print(b, type(b))  # Output: 123 <class 'int'>
         ```

       -
         ```
         # Float to int (truncates decimal)
         c = int(4.99)
         print(c)  # Output: 4
         ```

       -
         ```
         # Int to string
         d = str(100)
         print(d, type(d))  # Output: '100' <class 'str'>
         ```

       -
         ```
         # String to list
         e = list("hello")
         print(e)  # Output: ['h', 'e', 'l', 'l', 'o']
         ```

Control Flow

1. **if, elif, else Statements:** In Python, if, elif, and else statements are used for **conditional execution**. They allow your code to make decisions based on certain conditions.
   - **Syntax Overview**

     -
       ```
       if condition1:
           # code block if condition1 is True
       elif condition2:
           # code block if condition2 is True
       else:
           # code block if none of the above conditions are True
       ```
   - **Examples**
     - **Basic if statement**

       -
         ```
         x = 10
         if x > 5:
             print("x is greater than 5")
         ```
     - **Using if and else**

       -
         ```
         x = 3
         if x > 5:
             print("x is greater than 5")
         else:
             print("x is not greater than 5")
         ```
     - **Using if, elif, and else**

       -
         ```
         x = 5
         if x > 5:
             print("x is greater than 5")
         elif x == 5:
             print("x is exactly 5")
         else:
             print("x is less than 5")
         ```
     - **Multiple elif conditions**

       ```
       grade = 85
       if grade >= 90:
           print("A")
       elif grade >= 80:
           print("B")
       elif grade >= 70:
           print("C")
       else:
           print("F")
       ```

- **real-world example:** using **user input** for a simple login system: Example: Simple Login Check:
    - The user is prompted to enter their username and password.
    - The program checks:
        - If both are correct → ✅ Login successful.
        - If the username is wrong → ❌ Incorrect username.
        - If the password is wrong → ❌ Incorrect password.
    - It uses if, elif, and else to make decisions based on the input.

```
# Predefined correct username and password
correct_username = "admin"
correct_password = "1234"

# Get input from the user
username = input("Enter username: ")
password = input("Enter password: ")

# Check login credentials
if username == correct_username and password == correct_password:
   print("Login successful!")
elif username != correct_username:
   print("Incorrect username.")
elif password != correct_password:
   print("Incorrect password.")
else:
   print("Login failed.")
```

2. **for Loops:** for loop is used to **iterate** over a sequence like a list, string, or range of numbers.
    - Basic Syntax:
        - **for** variable **in** sequence**:**
        -    # do something
    - Examples
        - **Simple Example:**

            -
              ```
              fruits = ["apple", "banana", "cherry"]
              for fruit in fruits:
                 print("I like", fruit)
              ```

- **Looping through a string**

  -
    ```
    for letter in "hello":
      print(letter)
    ```
- **Using range() to loop a specific number of times**

  -
    ```
    for i in range(5):
      print(i)    //This prints: 0 1 2 3 4 (not 5)
    ```
- **Using enumerate():** enumerate() is a built-in Python function that adds a counter (index) to an iterable like a list or string. It lets you loop through both the index and the item at the same time.

  -
    ```
    fruits = ["apple", "banana", "cherry"]
    for index, fruit in enumerate(fruits):
      print(index, fruit)
    ```
    Example Without enumerate()
    ```
    fruits = ["apple", "banana", "cherry"]
    index = 0
    for fruit in fruits:
      print(index, fruit)
      index += 1
    ```

3. **while Loops:** In Python, a while loop is used to repeatedly execute a block of code **as long as a given condition is true**.
   - **Syntax:**

     -
       ```
       while condition:
         # code block
       ```
       - The condition is evaluated before each iteration.
       - The loop continues until the condition becomes False.
   - **Summary**
     - **Break:** Exit the loop
     - **Continue:** skip current iteration
     - **Pass:** Do nothing (placeholder only)
   - **Example:**

     -
       ```
       count = 0
       while count < 5:
         print("Count is:", count)
         count += 1
       ```
       ```
       Count is: 0
       Count is: 1
       Count is: 2
       Count is: 3
       Count is: 4
       ```

- **Infinite Loop**

  -
    ```
    while True:
      print("This will run forever unless you break it.")
      break  # used to stop the loop
    ```

- **Using break and continue**

  -
    ```
    x = 0
    while x < 5:
      x += 1
      if x == 3:
        continue  # skip this iteration
      if x == 5:
        break  # exit the loop
      print(x)
          1
          2
          4
    ```
- **Beware of Infinite Loops:** If the condition never becomes False, the loop will run forever. Make sure the loop has a condition that will eventually be false, or use break wisely.

  -
    ```
    # This is an infinite loop
    while 1 == 1:
      print("Looping forever")
    ```
- **Real-Life Example: ATM PIN Entry -** Imagine an ATM allows a user 3 attempts to enter the correct PIN before locking the account.

  -
    ```
    correct_pin = "1234"
    attempts = 0
    max_attempts = 3

    while attempts < max_attempts:
      entered_pin = input("Enter your PIN: ")
      if entered_pin == correct_pin:
        print("Access granted.")
        break
      else:
        print("Incorrect PIN.")
        attempts += 1

    if attempts == max_attempts:
      print("Account locked. Too many failed attempts.")
    ```

Functions

1) **Defining and Calling Functions**
   - Use the def keyword followed by the function name and parentheses ():
   - Calling a Function: You call a function by writing its name followed by parentheses:
   - **Example**

     -
       ```
       def greet():
          print("Hello, world!")
       greet()  #calling
       ```
       This defines a function named greet that prints Hello world!

2) **Function Parameters and Return Values**
   - **Functions with Parameters**

     -
       ```
       def greet(name):
          print(f"Hello, {name}!")
       greet("Alice")
       ```
   - **Function with Return Value (No Parameters)**

     -
       ```
       def get_greeting():
          return "Hello, world!"
       message = get_greeting()
       print(message)  # Output: Hello, world!
       ```
   - **Function with Parameters and Return Value**

     -
       ```
       def add(a, b):
          return a + b
       result = add(5, 3)
       print(result)  # Output: 8
       ```
   - **Multiple Parameters and Multiple Return Values**

     -
       ```
       def calculate(a, b):
          sum_ = a + b
          diff = a - b
          return sum_, diff
       s, d = calculate(10, 4)
       print("Sum:", s)      # Output: Sum: 14
       print("Difference:", d)  # Output: Difference: 6
       ```
   - **Default Parameter Values:** If a parameter isn't passed, the default is used.

     -
       ```
       def greet(name="Guest"):
          print(f"Hello, {name}!")
       greet()        # Output: Hello, Guest!
       greet("Sam")    # Output: Hello, Sam!
       ```

- **Keyword Arguments:** Arguments can be passed using the parameter names.

    -
      ```
      def introduce(name, age):
         print(f"{name} is {age} years old.")
      introduce(age=25, name="John")  # Output: John is 25 years old.
      ```
- **Variable Number of Arguments (*args and **kwargs)**
    - ***args –** multiple positional arguments:

        -
          ```
          def total(*args):
             return sum(args)
          print(total(1, 2, 3, 4))  # Output: 10
          ```
    - ****kwargs –** multiple keyword arguments:

        -
          ```
          def display_info(**kwargs):
             for key, value in kwargs.items():
                print(f"{key}: {value}")
          display_info(name="Alice", age=30)
          # Output:
          # name: Alice
          # age: 30
          ```
- **Functions Calling Other Functions**

    -
      ```
      def square(x):
         return x * x

      def double_square(y):
         return 2 * square(y)

      print(double_square(3))  # Output: 18
      ```

3) Scope (Global vs Local Variables)

- **Local Variables**
    - Declared inside a function.
    - Only accessible within that function.
    - Created when the function starts, destroyed when it ends.
    - **Example**

        -
          ```
          def greet():
             message = "Hello"  # local variable
             print(message)
          greet()
          # print(message)  # This would raise an error: NameError
          ```

- **Global Variables**
  - Declared outside of any function.
  - Can be accessed from **anywhere** in the script.
  - If used inside a function, they are **read-only** unless explicitly declared **global.**
  - **Example**

    -
      ```
      name = "Alice"  # global variable
      def greet():
          print("Hello", name)  # accesses global variable
      greet()
      ```
  - **Modifying Global Variables Inside Functions:** Use the global keyword to modify a global variable inside a function.

    -
      ```
      count = 0
      def increment():
          global count
          count += 1
      increment()
      print(count)  # Output: 1
      ```

Error Handling

1. In Python, **error handling** is done using the try, except, else, and finally blocks. These allow you to catch and respond to exceptions (errors) in a controlled way.
   - **Description of Each Clause**
     - **try**: Wrap code that might fail.
     - **except**: Handle specific or generic exceptions.
     - **else**: Runs only if the try block did **not** raise an exception.
     - **finally**: Always runs, useful for cleanup (like closing files or connections).
   - **Basic Structure**

     -
       ```
       try:
           # Code that might raise an exception
       ```

     -
       ```
       except SomeException:
           # Code that runs if the exception occurs
       ```

     -
       ```
       else:
           # Code that runs if no exception occurs
       ```

     -
       ```
       finally:
           # Code that always runs, no matter what
       ```

- **Examples**
  - **try + except:** Catch errors - Catch and handle specific errors (e.g., invalid input).

    -
      ```
      try:
         number = int(input("Enter a number: "))
      except ValueError:
         print("That's not a valid number.")
      ```
  - **try + except + else:** Run code if no error happens - Use else to run code only when no exception is raised.

    -
      ```
      try:
         number = int(input("Enter a number: "))
      except ValueError:
         print("That's not a valid number.")
      else:
         print(f"You entered: {number}")
      ```
  - **try + except + finally:** Run cleanup code regardless of error - Always run final code (e.g., close files, release resources).

    -
      ```
      try:
         number = int(input("Enter a number: "))
      except ValueError:
         print("That's not a valid number.")
      finally:
         print("Input attempt finished.")
      ```
  - **Full Structure: try + except + else + finally**

    -
      ```
      try:
         number = int(input("Enter a number: "))
         result = 10 / number
      except ValueError:
         print("Invalid input! Not a number.")
      except ZeroDivisionError:
         print("Cannot divide by zero.")
      else:
         print(f"Result is: {result}")
      finally:
         print("Done with operation.")
      ```

      Behavior:

      i. Input 5 → prints result and "Done with operation."
      ii. Input `abc` → catches `ValueError`.
      iii. Input `0` → catches `ZeroDivisionError`.
      iv. In all cases → `finally` runs

2. **Custom exceptions using the 'raise'**: The raise statement lets you manually trigger an exception. You can raise built-in exceptions like ValueError, or define your own custom exceptions.
   - **Raising a Built-in Exception:** If the user enters -5, Python raises: ValueError: Age cannot be negative!

     -
       ```
       age = int(input("Enter your age: "))
       if age < 0:
           raise ValueError("Age cannot be negative!")
       ```
   - **Defining a Custom Exception**

     -
       ```
       class NegativeAgeError(Exception):
           """Custom exception for negative ages."""
           Pass
       # This defines a new exception type. You can now raise it like any built-in error.
       ```
   - **Raising a Custom Exception**

     -
       ```
       class NegativeAgeError(Exception):
           pass
       age = int(input("Enter your age: "))
       if age < 0:
           raise NegativeAgeError("Custom: Age cannot be negative!")
       # Output (if -3 is entered):
       NegativeAgeError: Custom: Age cannot be negative!
       ```
   - **Handling Custom Exceptions with try/except**

     -
       ```
       class NegativeAgeError(Exception):
           pass
       try:
           age = int(input("Enter your age: "))
           if age < 0:
               raise NegativeAgeError("Custom: Age cannot be negative!")
       except NegativeAgeError as e:
           print(f"Caught error: {e}")
       ```

Data Structures

1. **lists**
   - **Creating a List**
     - my_list = [1, 2, 3, 4, 5]
     - mixed_list = [1, "hello", 3.14, True]
   - **Accessing List Items**
     - print(my_list[0])   # First item: 1
     - print(my_list[-1])   # Last item: 5

- **Modifying Lists**
  - my_list[1] = 20   # Changes 2 to 20
  - my_list.append(6)   # Adds 6 to the end
  - my_list.insert(2, 15)   # Inserts 15 at index 2
  - my_list.remove(4)   # Removes the first occurrence of 4
  - my_list.pop()   # Removes and returns the last item
- **List Operations**
  - new_list = my_list + [7, 8]   # Concatenation
    print(len(my_list))   # Length of list
- **Iterating Over a List**

  -
    ```
    for item in my_list:
        print(item)
    ```
- **Slicing Lists**
  - print(my_list[1:4])   # Items from index 1 to 3
  - print(my_list[:3])   # First 3 items
  - print(my_list[::-1])   # Reversed list
- **Useful List Methods**
  - my_list.sort()   # Sorts in place
  - my_list.reverse()   # Reverses in place
  - print(my_list.index(20)) # Finds index of 20
  - print(my_list.count(20)) # Counts occurrences of 20

2. **Tuples: tuple** is an **ordered, immutable** collection of elements. This means once a tuple is created, its contents **cannot be changed** (unlike lists). Tuples are commonly used to group related data.
   - **Creating a Tuple**
     - # Using parentheses
       my_tuple = (1, 2, 3)
     - # Without parentheses (optional)
       another_tuple = 4, 5, 6
     - # Single-element tuple (note the comma)
       single_element = (10,)
   - **Accessing Elements**
     - print(my_tuple[0]) # Output: 1
     - print(my_tuple[-1]) # Output: 3
   - **Slicing**
     - print(my_tuple[1:])   # Output: (2, 3)
     - print(my_tuple[:2])   # Output: (1, 2)

- **Tuple Packing and Unpacking**

    -
      ```
      # Packing
      ```

    -
      ```
      person = ("Alice", 30, "Engineer")

      # Unpacking
      name, age, job = person
      print(name)  # Output: Alice
      ```
- **Common Tuple Operations**

    -
      ```
      # Concatenation
      a = (1, 2)
      b = (3, 4)
      c = a + b  # (1, 2, 3, 4)

      # Repetition
      d = a * 3  # (1, 2, 1, 2, 1, 2)

      # Membership
      print(2 in a)  # True

      # Length
      len(a)  # 2

      # Count and Index
      (1, 2, 1, 3).count(1)   # 2
      (1, 2, 3).index(2)      # 1
      ```

3. **Dictionary:** a **dictionary** is a built-in data type that stores key-value pairs. Dictionaries are **unordered**, **mutable**, and **indexed by keys**, which can be strings, numbers, or even tuples (if immutable).
    - **Basic Syntax**

        -
          ```
          # Creating a dictionary
          my_dict = {
             "name": "Alice",
             "age": 25,
             "is_hacker": True
          }
          ```
    - **Common Operations**

        -
          ```
          # Accessing a value
          print(my_dict["name"])  # Output: Alice

          # Using .get() (avoids error if key is missing)
          print(my_dict.get("email", "Not found"))  # Output: Not found
          ```

```
# Adding or updating a value
my_dict["email"] = "alice@example.com"

# Removing a key-value pair
del my_dict["age"]  # or use .pop()
```

- **Iterating Over a Dictionary**

  -
    ```
    for key in my_dict:
        print(key, my_dict[key])

    # or
    for key, value in my_dict.items():
        print(f"{key}: {value}")
    ```
- **Useful Methods**
  - my_dict.keys()   # returns all keys
  - my_dict.values()   # returns all values
  - my_dict.items()   # returns all (key, value) pairs
  - my_dict.clear()   # removes all items
- **Example Use Case**

  -
    ```
    # Count frequency of characters in a string
    s = "hacker"
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    print(freq)  # Output: {'h': 1, 'a': 1, 'c': 1, 'k': 1, 'e': 1, 'r': 1}
    ```

4. **Sets:** a **set** is an **unordered collection** of **unique, immutable elements**. Sets are useful when you want to store items without duplicates and perform operations like union, intersection, and difference.
   - **Basic Properties**
     - **Unordered:** Items have no index or order.
     - **Mutable**: You can add/remove items.
     - **Unique elements**: Duplicates are automatically removed.
   - **Creating a Set**

     -
       ```
       my_set = {1, 2, 3}
       empty_set = set()  # NOT {} — this creates a dictionary
       ```
   - **Common Set Methods**
     - s = {1, 2, 3}
     - s.add(4)   # {1, 2, 3, 4}
     - s.remove(2)   # {1, 3, 4}
     - s.discard(5)   # No error if 5 isn't present
     - s.pop()   # Removes a random element
     - s.clear()   # Empties the set

- **Rules to Identify Set or Dictionary:** The syntax {} can be confusing in Python because it's used for both **dictionaries** and **sets**, but there's a key difference:
  - If the elements look like **key-value pairs** (key: value), it's a **dictionary**.
  - If the elements are **single values** (no colons), it's a **set**.
  - If you do {} without anything inside, it creates an **empty dictionary**, not a set.

  -
    ```
    s = {1, 2, 3}
    print(type(s))     # <class 'set'>

    d = {"a": 1}
    print(type(d))     # <class 'dict'>

    empty = {}
    print(type(empty)) # <class 'dict'>

    empty_set = set()
    print(type(empty_set))  # <class 'set'>
    ```
- **Set Operations**

  -
    ```
    a = {1, 2, 3}
    b = {3, 4, 5}
    a | b      # Union => {1, 2, 3, 4, 5}
    a & b      # Intersection => {3}
    a - b      # Difference => {1, 2}
    a ^ b      # Symmetric Difference => {1, 2, 4, 5}
    ```
- **Useful Functions**
  - `len(s)        # Number of items`
  - `3 in s        # Check if 3 is in the set`

5. **Nested data structures** (like dictionaries inside lists, lists inside dictionaries, or deeper nesting)
   - **List inside a List**

     -
       ```
       data = [[1, 2], [3, 4], [5, 6]]
       for sublist in data:
           for item in sublist:
               print(item)
       ```
   - **List of Dictionaries**

     -
       ```
       data = [
           {"name": "Alice", "age": 25},
           {"name": "Bob", "age": 30}
       ]
       for person in data:
           for key, value in person.items():
               print(f"{key}: {value}")
       ```

- **Dictionary with List Values**

  -
    ```
    data = {
      "fruits": ["apple", "banana"],
      "vegetables": ["carrot", "lettuce"]
    }
    for category, items in data.items():
      print(f"{category}:")
      for item in items:
        print(f"  {item}")
    ```
- **Dictionary inside a Dictionary**

  -
    ```
    data = {
      "server1": {"ip": "192.168.1.1", "status": "active"},
      "server2": {"ip": "192.168.1.2", "status": "inactive"}
    }
    for server, info in data.items():
      print(f"{server}:")
      for key, value in info.items():
        print(f"  {key} = {value}")
    ```
- **List of Dicts with Nested Dicts**

  -
    ```
    data = [
      {
        "user": "alice",
        "info": {"email": "alice@example.com", "active": True}
      },
      {
        "user": "bob",
        "info": {"email": "bob@example.com", "active": False}
      }
    ]

    for item in data:
      print(item["user"])
      for key, value in item["info"].items():
        print(f"  {key}: {value}")
    ```

OOP

1. **constructor __init__**: The constructor method to initialize attributes when creating an object.
   - The __init__ method is a **special method** in Python. It's called **automatically when you create a new object** from a class. Its main job is to **initialize (set up) the object's attributes** (the data it holds).
   - Think of __init__ like this:
     - When you build a new car, the __init__ method is like the assembly process that gives it its color, engine, and wheels.
     - When you create an object, __init__ sets up its initial values.
   - **Syntax:**

     -
       ```
       def __init__(self, parameter1, parameter2, ...):
           self.attribute1 = parameter1
           self.attribute2 = parameter2

       #self refers to the current object being created.
       #parameter1, parameter2, etc., are values you pass when creating the object.
       #Inside __init__, you use self.attribute = value to store those values in the object.
       ```
2. **Classes and Objects:** A class is like a blueprint for creating objects. An object is an instance of a class. The class defines the properties and behaviors, while the object is an actual entity created using that class.
   - **Attributes**: Variables that belong to the object (instance variables).
   - **Methods**: Functions that belong to the class and define the object's behavior.
   - In the example below, name and age are attributes, and bark() is a method.
   - Example

     -
       ```
       # Define a class
       class Dog:
           # Define attributes and behaviors (methods) in the class
           def __init__(self, name, age):
               self.name = name  # Attribute: name of the dog
               self.age = age    # Attribute: age of the dog

           def bark(self):
               print(f"{self.name} says woof!")

       # Create an object (instance of the class)
       my_dog = Dog("Buddy", 3)

       # Access object attributes
       ```

```
print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 3

# Call the object method
my_dog.bark()       # Output: Buddy says woof!
```

3. **Inheritance:** Inheritance allows a class to inherit attributes and methods from another class. The new class (child class) can extend or modify the behavior of the parent class.
   - **class** father: #declaring class with name father
     class son (father): #now class son inherits all class father's activities
   - **Class** A:
     Class B: #let's say there is no relationship b/w class A and B
     Class C (A,B): # class C, inherits both classes A and B
   - **Example**

     -
       ```
       # Parent class
       class Animal:
           def __init__(self, name):
               self.name = name

           def speak(self):
               print(f"{self.name} makes a sound.")

       # Child class
       class Dog(Animal):
           def __init__(self, name, breed):
               super().__init__(name)  # Call the parent constructor
               self.breed = breed

           def bark(self):
               print(f"{self.name} says woof!")

       # Create an object of the child class
       my_dog = Dog("Buddy", "Golden Retriever")
       my_dog.speak()  # Inherited from Animal class
       my_dog.bark()   # Defined in Dog class
       ```

       #super(): A function that allows the child class to call methods from the parent class.

File and Log Handling

1. **External Files:**
   - **Syntax*:***
     - **variable = open (" file_name.extension " , " reason ")**
     - The reason will be one of these
       - ✓ **"r" =** read, this means I am opening the file for reading
       - ✓ **"w "=** write, when you use "w" in existed file it will overwrite, this means it will delete the previous file and write new things, we can also use "w" to create a file
       - ✓ **"a "=** append, adding something at the end of the file without deleting.
       - ✓ **"r+ "=** read and write
     - **Creating file "w"**
       - ✓ student_file = **open** ('student.txt', **'w'**)
         student_file.**write**(' New student record file \n ')
         student_file.**close**()
     - **Reading files "r"**
       - ✓ student_file = **open** ('student.txt', **'r'**)
       - ✓ print(student_file.**readable()**) #this function returns True or False, if the file is readable it will return True else False
       - ✓ print(student_file.**read()**) #this function reads and prints all the data in the file
       - ✓ print(student_file.**readlines()[1]**) // reading specific line if you only want, like this example I am only reading and printing line index [1]
       - ✓ student_file.**close**()
     - **Writing files "w":** Let's say we have student.txt (file) and their record of the student so if you want to overwrite (deleting the previous record and writing new one) use "w" open and then overwrite examples
       - ✓ student_file = **open** ('student.txt', **'w'**)
         student_file.**write**(' student name is adan \n ')
         student_file.**close**()
     - **Append "a":** Append is adding something to an existed file without deleting anything , and append will add at the last
       - ✓ student_file = **open** ('student.txt', **'a'**)
       - ✓ student_file.**write**(' student age 26 \n ')'
       - ✓ student_file.**close**()

2. **working with file paths**
   - **Working with file paths in Python** means managing how your Python program locates and interacts with files and directories in the file system—like opening, reading, writing, or checking if a file exists.
   - This is especially important when:
     - Writing cross-platform code (Linux, Windows, macOS).
     - Automating file operations (like in cybersecurity scripts or log analysis).
     - Navigating through directories or generating reports.
   - The best and most modern way to handle paths is using the pathlib module, introduced in Python 3.4.
   - **Import the Module**:

     ```
     from pathlib import Path
     ```
   - **Locating a File or Directory:** This means pointing your program to the location of a file or folder using a *path*.

     -
       ```
       from pathlib import Path

       # Pointing to a file path (relative path)
       file_path = Path("data/example.txt")

       # Pointing to a directory
       folder_path = Path("data/logs")
       ```
     - If the path is *absolute*, it will look like this:

       ```
       file_path = Path("/home/user/data/example.txt")
       ```
   - **Checking If a File or Directory Exists:** Before opening or modifying a file, it's important to make sure it exists.

     -
       ```
       from pathlib import Path

       file_path = Path("data/example.txt")

       if file_path.exists():
           print("The file exists.")
       else:
           print("The file does NOT exist.")
       ```
   - **Checking If It's a File or Directory:** To know what you're working with—file or folder.

     -
       ```
       if file_path.is_file():
           print("It's a file.")

       if file_path.is_dir():
           print("It's a folder.")
       ```

- **Appending to a File (Add Without Overwriting):** pathlib doesn't support append directly, so use open():

  -
    ```
    with file_path.open("a") as file:
        file.write("\nAnother line.")
    ```
- **Getting File Details:** You can extract file name, extension, parent folder, etc.
  - print(file_path.name)   # example.txt
  - print(file_path.stem)   # example
  - print(file_path.suffix)   # .txt
  - print(file_path.parent)   # data/
- **Finding Files in a Folder:** List files or search for specific types (e.g., .log files).

  -
    ```
    logs_folder = Path("data/logs")
    for log_file in logs_folder.glob("*.log"):
        print(log_file)
    ```
    - ✓ #Use .rglob("*.log") to search **recursively** in subfolders too.
- **Creating a Directory:** If it doesn't exist, create it.

  -
    ```
    new_folder = Path("output/reports")
    new_folder.mkdir(parents=True, exist_ok=True)
    ```
- **Delete a file**

  -
    ```
    if file_path.exists():
        file_path.unlink()
    ```
- Delete an empty folder)

  -
    ```
    if new_folder.exists():
        new_folder.rmdir()
    ```

3. **handling large log files**
   - Large log files (like .log, .txt, .pcap, etc.) can be several GBs in size. You **can't load the entire file into memory** at once, or it will crash your program. So you need to:
     - Read logs **line by line**
     - Filter or extract important information efficiently
     - Avoid memory overload
   - **Reading Large Log Files Line by Line:** Use a for loop with open(), which is memory efficient (This reads one line at a time — good for logs with **millions of lines**.)

     -
       ```
       with open("server.log", "r") as log_file:
           for line in log_file:
               print(line.strip())  # .strip() removes extra newlines
       ```
   - **Filter Specific Lines (e.g., Errors Only) :** This only prints lines that contain "ERROR" — useful for parsing alerts or issues.

     -
       ```
       with open("server.log", "r") as log_file:
           for line in log_file:
               if "ERROR" in line:
                   print(line.strip())
       ```

- **Write Filtered Results to a New File:** This creates a **smaller file** with only the important parts — good for forensic analysis or backups.

  -
    ```
    with open("server.log", "r") as source, open("errors.log", "w") as target:
        for line in source:
            if "ERROR" in line:
                target.write(line)
    ```
- **Count Specific Events**: Count how many times "failed login" appears.

  -
    ```
    count = 0
    with open("auth.log", "r") as log_file:
        for line in log_file:
            if "Failed password" in line:
                count += 1
    print(f"Failed logins: {count}")
    ```
- **Using Generators:** To process very large files **lazily**, you can use generator functions. This approach **doesn't store all lines in memory** — it's ideal for big data logs.

  -
    ```
    def error_lines(filepath):
        with open(filepath) as f:
            for line in f:
                if "ERROR" in line:
                    yield line

    for error in error_lines("server.log"):
        print(error.strip())
    ```
- **Handle Gzipped Log Files:** Sometimes logs are compressed as .gz. You can read them like this:

  -
    ```
    import gzip

    with gzip.open("server.log.gz", "rt") as log_file:
        for line in log_file:
            if "ERROR" in line:
                print(line.strip())
    ```

JSON and CSV Handling

1. **JSON (JavaScript Object Notation)** is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It's used primarily to exchange data between a server and a client, especially in web applications. JSON structures data as key-value pairs, making it both simple and flexible for various programming languages, including Python.

2. **JSON looks like this:**
```
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
# "name", "age", and "city" are keys.
#"John", 30, and "New York" are the corresponding values.
```
3. **Why Do We Need to Use JSON in Python?**
   - **Data Exchange**: JSON is a common format for transferring data between servers and clients. Python, being a versatile language, can handle JSON easily, making it ideal for interacting with web APIs and other services.
   - **Readability**: JSON is human-readable and easy to understand, which makes it great for logging and debugging.
   - **Interoperability**: Python uses the json module to parse and generate JSON data, which allows it to work seamlessly with many web technologies and systems that rely on JSON (e.g., APIs, web scraping, data storage).
   - **Lightweight**: Compared to other data formats like XML, JSON is more compact, which helps improve the performance of applications, especially in situations with large datasets.
4. Python provides a built-in library called json to work with JSON data. Here's how you can use it:
5. **Loading JSON Data (Parse JSON):** To convert a JSON string into a Python dictionary, you use json.loads() (load string):

   -
     ```
     import json

     # JSON string
     json_string = '{"name": "John", "age": 30, "city": "New York"}'

     # Parse JSON string into Python dictionary
     data = json.loads(json_string)

     print(data)
     print(type(data))

     #This converts the json_string into a Python dictionary, allowing you to work
     with the data as if it were any other dictionary.
     ```
6. **Writing JSON Data (Convert Python Object to JSON):** To convert a Python dictionary into a JSON string, you use json.dumps() (dump string):

   -
     ```
     import json
     # Python dictionary
     data = {
     ```

```
  "name": "John",
  "age": 30,
  "city": "New York"
}

# Convert Python dictionary to JSON string
json_string = json.dumps(data)

print(json_string)
print(type(json_string))
```

7. **Reading JSON from a File (json.load())**

   -
     ```
     # Reading from a file
     with open('data.json', 'r') as f:
         data = json.load(f)
     print(data["name"])  # Output: Alice
     ```
8. **Parsing JSON logs**
   - **Parsing** means:
     - **Reading** the JSON log (from a file or string),
     - **Converting** it into a Python dictionary or object,
     - **Accessing** its fields for analysis, filtering, or alerting.
   - **Why Parse JSON Logs?**
     - To detect suspicious activity (e.g., multiple failed logins).
     - To extract fields like IPs, usernames, timestamps.
     - To generate reports or alerts.
   - **Python Example: Parsing a JSON Log File:** Suppose you have a log file with one JSON object per line:
     - {"timestamp": "2025-05-06T10:15:30Z", "event": "login", "user": "alice", "ip": "192.168.1.10"}
     - {"timestamp": "2025-05-06T10:17:00Z", "event": "failed_login", "user": "bob", "ip": "10.0.0.5"}

       Here's how to parse it:
       - ✓
         ```
         import json

         with open("logs.json", "r") as f:
             for line in f:
                 log = json.loads(line)  # Parse the JSON string into a dict
                 if log["event"] == "failed_login":
                     print(f"Failed login by {log['user']} from {log['ip']}")
         ```

9. **Reading and writing CSV:** CSV stands for **Comma-Separated Values**. It is a simple file format used for storing tabular data, such as a spreadsheet or database. Each line in a CSV file represents a row of data, and the columns are separated by commas (or sometimes other delimiters like semicolons or tabs). It’s commonly used for data exchange between different applications, such as exporting data from databases, spreadsheets, or other systems.
   - **Example of a CSV File:**
     - Name,Age,Location
     - John,25,New York
     - Alice,30,San Francisco
     - Bob,22,Chicago
       - ✓ The first line defines the column headers: Name, Age, and Location.
       - ✓ The subsequent lines contain the data for each person: their name, age, and location.
   - **Using the csv Module in Python:** Python's csv module allows easy reading from and writing to CSV files. Here's how you can work with CSV files using Python.
   - **Reading a CSV file**: You can read a CSV file by using the csv.reader() function, which returns an iterator that will iterate over the lines in the specified CSV file.

     -
       ```
       import csv

       # Open the CSV file in read mode
       with open('data.csv', mode='r') as file:
           csv_reader = csv.reader(file)

           # Loop through the rows and print them
           for row in csv_reader:
               print(row)
       ```
       - ✓ ['Name', 'Age', 'Location']
       - ✓ ['John', '25', 'New York']
       - ✓ ['Alice', '30', 'San Francisco']
       - ✓ ['Bob', '22', 'Chicago']
   - **Writing to a CSV file**: You can write data to a CSV file using the csv.writer() function. This allows you to create a new CSV file or overwrite an existing one.

     -
       ```
       import csv

       # Data to write to the CSV file
       data = [
           ['Name', 'Age', 'Location'],
       ```

```
    ['John', 25, 'New York'],
    ['Alice', 30, 'San Francisco'],
    ['Bob', 22, 'Chicago']
]

# Open the CSV file in write mode
with open('new_data.csv', mode='w', newline='') as file:
    csv_writer = csv.writer(file)

    # Write the data to the file
    csv_writer.writerows(data)
```

- ✓ This will create a CSV file (new_data.csv) with the following content:

```
Name,Age,Location
John,25,New York
Alice,30,San Francisco
Bob,22,Chicago
```

Regular expressions

re.search() and **re.findall()**: In Python, the re.search() and re.findall() functions are part of the re (regular expressions) module, which is essential for pattern matching in text. These functions are particularly useful for tasks in cybersecurity, such as searching for patterns in log files, URLs, IP addresses, or other network-related data.

- **re.search():** The re.search() function scans a string for a pattern and returns the **first match** it finds. If it doesn't find any match, it returns None.
  - **Syntax**:
    - re.search(pattern, string)
  - **Parameters**:
    - **pattern:** The regular expression you want to match.
    - **string:** The string you want to search in.
  - **Return**: A match object if the pattern is found, otherwise None.
  - **Example: Searching for an IP address:** Let's say you have a log file and you want to find the first IP address in the text:

    -
      ```
      import re

      # Sample log data
      log_data = "Connection from 192.168.0.1 at 10:15:00"

      # Regex pattern for an IP address
      ```

```
pattern = r"\b\d{1,3}(\.\d{1,3}){3}\b"

# Search for the pattern
match = re.search(pattern, log_data)

if match:
    print("Found IP address:", match.group())
else:
    print("No IP address found")
```

        - **Explanation**:
            - The regex pattern r"\b\d{1,3}(\.\d{1,3}){3}\b" matches an IP address.
            - match.group() retrieves the matched string (IP address).
    - The regex pattern r"\b\d{1,3}(\.\d{1,3}){3}\b" is designed to match an **IP address** in the form of four sets of 1 to 3 digits, separated by periods (e.g., 192.168.0.1).
        - **\b**: Word boundary (ensures the match is not part of a larger word).
        - **\d{1,3}**: Matches 1 to 3 digits (for each section of the IP address).
        - **(\.\d{1,3}){3}**: Matches a period (\.) followed by 1 to 3 digits, repeated 3 times (for the 3 periods separating the sections).
        - **\b**: Word boundary (ensures the match is a complete IP address).
        - **Example matches**:
            - 192.168.0.1
            - 10.0.0.1
            - 172.16.0.5
        - It will **not** match:
            - 192.168.0.256 (invalid IP because 256 is over 255)
            - abc192.168.0.1xyz (because it's not a complete word).
- **re.findall():** The re.findall() function finds **all non-overlapping matches** of the pattern in the string and returns them as a list.
    - **Syntax**:
        - re.findall(pattern, string)
    - **Parameters**:
        - **pattern:** The regular expression to search for.
        - **string:** The string to search in.
    - **Return**: A list of all matches.

- **Example: Finding all IP addresses in a log file -** If you want to find all IP addresses in a log file, you can use re.findall()

    -
      ```
      import re

      # Sample log data
      log_data = """
      Connection from 192.168.0.1 at 10:15:00
      Connection from 10.0.0.1 at 11:20:00
      Connection from 172.16.0.1 at 12:30:00
      """

      # Regex pattern for an IP address
      pattern = r"\b\d{1,3}(\.\d{1,3}){3}\b"

      # Find all IP addresses
      ips = re.findall(pattern, log_data)

      print("Found IP addresses:", ips)
      ```

    Explanation:

    re.findall() returns all IP addresses found in the log_data.

    The output would be a list of all IP addresses in the log data.

- **How these functions are useful in cybersecurity**
    - **Log Analysis**: Extract IP addresses, user-agent strings, or other relevant data from server or application logs.
    - **Pattern Detection**: Look for specific patterns like suspicious URLs, IPs, or commands in network traffic captures (e.g., from Wireshark or Zeek).
    - **Malware Analysis**: Detect patterns related to command-and-control communications or file names used by malware.
    - **Phishing Detection**: Identify potentially dangerous links by checking URL patterns.

- **Writing Patterns to Match Log or Payload Content:**
    - **Match Email Addresses:** This script searches the log for an email address format and prints it if found. Useful for extracting user credentials or indicators in logs.

        -
          ```
          import re
          log = "User test@example.com failed to login from 192.168.1.5"
          pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"
          match = re.search(pattern, log)
          if match:
            print("Found email:", match.group())
          ```

- **Match URLs (Phishing Detection):** This script finds and lists all URLs in the input string. Great for detecting phishing links in payloads or emails.

    -
      ```
      log = "Malicious link: http://bad.com/path?query=123"
      pattern = r"http[s]?://[^\s<>\"']+"
      urls = re.findall(pattern, log)
      print("Found URLs:", urls)
      ```
- **Match Timestamps:** Finds timestamps in YYYY-MM-DD HH:MM:SS format. Helps correlate events in logs during an investigation.

    -
      ```
      log = "Login failed at 2025-05-06 14:22:01 from 10.0.0.2"

      pattern = r"\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}"
      timestamp = re.search(pattern, log)
      if timestamp:
          print("Found timestamp:", timestamp.group())
      ```
- **Match HTTP Methods:** Extracts HTTP methods (e.g., GET, POST). Useful when analyzing web server logs for request behavior.

    -
      ```
      http_request = "POST /login HTTP/1.1"

      pattern = r"\b(GET|POST|PUT|DELETE|HEAD|OPTIONS)\b"
      method = re.search(pattern, http_request)
      if method:
          print("Found HTTP method:", method.group())
      ```
- **Match Failed Login Attempts:** Detects failed SSH login attempts and extracts the attacker's IP address. Very useful in brute-force detection.

    -
      ```
      log = "ERROR: Failed password for invalid user root from
      192.168.0.101"

      pattern = r"Failed password for .* from (\b\d{1,3}(\.\d{1,3}){3}\b)"
      match = re.search(pattern, log)
      if match:
          print("Failed login from IP:", match.group(1))
      ```

Using External Libraries

1. **Installing packages with pip:**

- Installing packages with pip in Python is how you add external libraries (modules) that aren't included in the standard library. Here's a quick guide to using pip
- Basic Installation Command
    - ***pip install package_name***
- **For example:** It installs the ***requests*** library, which lets Python send HTTP requests easily
    - pip install requests
- **Common Variants**
    - **Install latest version:** pip install somepackage
    - **Install specific version:** pip install somepackage==1.2.3
    - **Update to latest version:** pip install --upgrade somepackage

    - **Remove package:** pip uninstall somepackage
    - **Show all installed packages:** pip list
    - **Output installed packages (useful for requirements.tx)t:** pip freeze
- **Installing from requirements.txt:** pip install -r requirements.txt

2. **importing modules:** In Python, importing modules allows you to reuse code written in other files or packages. This is essential for organizing programs and using built-in or third-party functionality.
    - **Import the whole module**
        - import math
          print(math.sqrt(16)) # Output: 4.0
    - **Import with an alias**
        - import numpy as np   #use np instead of typing numpy everytime
          print(np.array([1, 2, 3]))
    - **Import specific functions or variables**
        - from math import sqrt, pi
          print(sqrt(25)) # Output: 5.0
          print(pi)   # Output: 3.141592653589793
    - **Import all names (not recommended):** Avoid this style in larger scripts—can cause name conflicts.
        - from math import *
          print(sin(pi / 2)) # Output: 1.0
    - **Importing Your Own Files:** instead of writing all codes in one file it's better to separate them into different files then link them all together like HTML
        - Create two python files first example, ***main.py*** and ***calc.p***
        - In the ***calc.py*** write some lines of code example

            -
              ```
              Def add (a,b):
                  return a+b
              Def sub (a,b):
                  return a-b
              Def multiple  (a,b):
                return a*b
              ```
        - In the second file ***main.py***

            -
              ```
              from calc import *   #linking main.py to calc.py
              a,b = 1,2
              C = add(a,b)
              #passing two variables to another module or file File and Log
              Handling
              ```

3. **Common modules os, sys, time, and random**
   - **os Module:** The os module provides a way of interacting with the operating system. It allows you to perform tasks like manipulating file paths, reading environment variables, and managing processes.
     - **Key Functions:**
       - **os.getcwd():** Get the current working directory.
       - **os.listdir():** List files in a directory.
       - **os.rename():** Rename a file.
       - **os.remove():** Remove a file.
     - **Example:**

       -
         ```
         import os

         # Get current working directory
         current_directory = os.getcwd()
         print(f"Current Directory: {current_directory}")

         # List files in the current directory
         files = os.listdir(current_directory)
         print(f"Files in current directory: {files}")

         # Create a new directory
         os.mkdir("new_directory")

         # Rename the directory
         os.rename("new_directory", "renamed_directory")

         # Remove the directory
         os.rmdir("renamed_directory")
         ```
   - **sys Module:** The sys module allows you to interact with the interpreter and system-specific parameters. It can help with managing command-line arguments, exiting the program, and modifying the system path.
     - **Key Functions:**
       - **sys.argv:** A list of command-line arguments passed to the script.
       - **sys.exit():** Exit the program.
       - **sys.path:** A list of directories that the interpreter searches for modules.
     - **Example:**

       -
         ```
         import sys

         # Check command-line arguments
         if len(sys.argv) > 1:
         ```

```
  print(f"Arguments: {sys.argv}")
else:
  print("No arguments provided.")

# Exit the program
sys.exit("Exiting the program now.")
```

- **time Module:** The time module provides functions for working with time, including measuring execution time, pausing execution, and formatting dates/times.
  - **Key Functions:**
    - **time.sleep():** Pause execution for a specified number of seconds.
    - **time.time():** Get the current time in seconds since the epoch (1970).
    - **time.strftime():** Format a time object as a string.
  - **Example:**

    -
      ```
      import time

      # Pause for 2 seconds
      print("Starting the pause...")
      time.sleep(2)
      print("Pause complete.")

      # Measure execution time
      start_time = time.time()
      time.sleep(1)
      end_time = time.time()
      execution_time = end_time - start_time
      print(f"Execution time: {execution_time} seconds")

      # Format current time
      current_time = time.strftime("%Y-%m-%d %H:%M:%S",
      time.gmtime())
      print(f"Current time: {current_time}")
      ```
- **random Module:** The random module provides functions for generating random numbers, shuffling items, and choosing random elements from sequences.
  - **Key Functions:**
    - **random.randint():** Generate a random integer in a specified range.
    - **random.choice():** Choose a random element from a list.
    - **random.shuffle():** Shuffle a list randomly.

- **Example**
    - import random

      ```
      # Generate a random integer between 1 and 10
      random_int = random.randint(1, 10)
      print(f"Random integer: {random_int}")

      # Choose a random item from a list
      fruits = ["apple", "banana", "cherry"]
      random_fruit = random.choice(fruits)
      print(f"Random fruit: {random_fruit}")

      # Shuffle a list
      random.shuffle(fruits)
      print(f"Shuffled list: {fruits}")
      ```

- **Summary of Key Points:**
  - **os**: For interacting with the operating system (files, directories, processes).
  - **sys**: For interacting with the Python interpreter (command-line arguments, exiting the program).
  - **time**: For working with time (pausing execution, measuring performance, formatting times).
  - **random**: For generating random numbers and making random selections (useful for simulations, games, etc.).

Debugging and Logging

1. **Using *pdb* for Step-by-Step Debugging:** Using **pdb** (Python Debugger) for step-by-step debugging is a powerful way to inspect and control your code execution. Here's a practical guide on how to use it effectively:
   - **Insert pdb in Your Code:** To start debugging at a specific line, insert:
     - import pdb; pdb.set_trace() #This will pause execution and drop you into the interactive debugger.
   - **Basic pdb Commands**
     - **N:** Next line (step **over** function calls)
     - **S:** Step **into** the function
     - **C:** Continue execution until the next breakpoint
     - **Q:** Quit debugger
     - **L:** List code around the current line
     - **P:** Print the value of a variable (p var_name)
     - **B:** Set a breakpoint (b 12 sets it at line 12
     - **C1:** Clear breakpoints (cl or cl 12)
     - **!:** Run a Python command (e.g., !x + 1)

    - **H:** Show help
- **Example**

    -
      ```
      def add(x, y):
        result = x + y
        return result

      def main():
        a = 5
        b = 10
        import pdb; pdb.set_trace()  # <-- Debugging starts here
        c = add(a, b)
        print(c)

      main()

      #When you run this, Python will stop at pdb.set_trace() and allow you to inspect variables and step through code.
      ```

2. **Logging with logging Module (vs. print):** Using the logging module instead of print() in Python is considered best practice for serious or production-level code. Here's a breakdown of **why and how** you should use logging.
    - **print() vs. logging**

Feature	**print()**	**logging Module**
Output Destination	Always stdout	stdout, file, network, etc.
Message Level	No level (just prints)	Supports DEBUG, INFO, WARNING, ERROR
Filtering	Not supported	Yes (via log level)
Configurable Format	Manual formatting only	Built-in formatting options
Scalable for Production	❌ Not suitable	✅ Standard for production
Timestamp / Metadata	Must add manually	Automatic with configuration

- **Basic Example**

  -
    ```
    import logging

    logging.basicConfig(level=logging.INFO)
    logging.info("This is an info message.")

    #Output:
    INFO:root:This is an info message.
    ```
- **Logging Levels:**
  - **DEBUG:** Detailed diagnostic info
  - **INFO:** General information (e.g., progress)
  - **WARNING:** Something unexpected but not fatal
  - **ERROR:** A serious problem, part of app failed
  - **CRITICAL:** Severe problem, app may crash
- More Configurable Logging

  -
    ```
    logging.basicConfig(
      level=logging.DEBUG,
      format='%(asctime)s - %(levelname)s - %(message)s',
      filename='app.log',  # log to file
      filemode='w'        # overwrite file each run
    )

    logging.debug("Debug message")
    logging.info("Info message")
    logging.warning("Warning!")
    logging.error("Error occurred")
    logging.critical("Critical issue")
    ```

Virtual environments

Creating Isolated Environments with venv: Creating isolated environments with venv in Python is an essential practice, especially in cybersecurity, scripting, and development work, because it keeps dependencies organized and prevents conflicts between projects.

- It’s a tool in Python that lets you create a **mini environment** just for your project
- You can install packages **without affecting** other projects.
- Everything stays clean and organized.

- **For Windows**
  - **Go to your project folder:** Open Command Prompt (or PowerShell), then:
    - cd my_project
  - **Create a virtual environment:**
    - python -m venv env   #env is the name of the environment folder. You can name it anything (e.g., .venv, venv, myenv)
  - **Activate the environment:**
    - env\Scripts\activate
  - **You'll now see:**
    - (env) C:\Users\You\my_project>
  - **Install packages:**
    - pip install requests
  - **Deactivate when done:**
    - Deactivate
- **For Linux/macOS**
  - **Go to your project folder:**
    - cd my_project
  - **Create a virtual environment:**
    - python3 -m venv env
  - **Activate the environment:**
    - source env/bin/activate
  - **You'll now see:**
    - (env) $
  - **Install packages:**
    - pip install requests
  - **Deactivate when done:**

Script Structure and Best Practices

Structuring your Python scripts properly is essential for writing **maintainable, reusable, and testable code**, especially in cybersecurity and hacking where you'll often reuse scripts or modules in multiple contexts. Here's a clear guide to **Python script structure and best practices**, including how and why to use the if __name__ == "__main__": guard.

- **Ideal Python Script Structure**
  - #!/usr/bin/env python3

```
"""
Module Docstring:
A brief description of what this script does.
Author: Your Name
```

```
Date: YYYY-MM-DD
"""

import sys
import os
import argparse
import logging

# === Constants ===
VERSION = "1.0"

# === Functions ===
def do_something_important():
    """Performs the main task."""
    pass  # Replace with real logic

# === Classes (if needed) ===
class Tool:
    def __init__(self):
        pass

    def run(self):
        do_something_important()

# === Main Function ===
def main():
    """Main execution logic."""
    parser = argparse.ArgumentParser(description="Describe your script.")
    parser.add_argument("-v", "--verbose", action="store_true", help="Enable
verbose output")
    args = parser.parse_args()

    if args.verbose:
        logging.basicConfig(level=logging.DEBUG)

    logging.debug("Verbose mode enabled")
    do_something_important()

# === Entry Point ===
if __name__ == "__main__":
    main()
```

- **Why Use if __name__ == "__main__":** When you run a Python file, Python sets the special variable __name__:
    - If the file is run directly → __name__ == "__main__"
    - If the file is imported as a module → __name__ == "filename"
- **Purpose of the Guard**
    - Ensures that **main()** runs only when the script is executed directly.
    - Prevents unintended execution when the file is **imported into another script**.
        - **file: tools.py**
            -
              ```
              def say_hello():
                 print("Hello!")

              def main():
                 print("Running main function")

              if __name__ == "__main__":
                 main()
              ```
        - **file: app.py**
            -
              ```
              import tools
              tools.say_hello()

              #say_hello() runs.
              #main() does not run, because tools.py was imported.
              #So you can use say_hello() without running the full program
              in tools.py
              ```
- **Best Practices Checklist**
    - **Imports:** Standard library first, then 3rd-party, then your own modules. Alphabetical or logical order.
    - **Docstrings:** Add a module-level and function-level docstring. Use """Triple quotes""".
    - **Main Function:** Wrap execution logic in main() to keep global scope clean.
    - **Argument Parsing:** Use argparse to handle CLI arguments instead of sys.argv.
    - **Logging:** Use logging instead of print() for real scripts. It's more flexible.
    - **Constants:** Use UPPER_CASE for constants.
    - **Testing:** Structure code so functions can be tested independently.
    - **Naming:** Use snake_case for functions/variables and CamelCase for classes.
    - **Modularization:** Break complex scripts into multiple files/modules if needed.
    - **Virtual Environments:** Use venv to isolate dependencies.
- **Example Use Case: Cybersecurity Script**

```
# port_scanner.py
def scan_ports(target_ip):
   """Scans ports on a given target IP."""
```

```
        pass

    def main():
        import argparse
        parser = argparse.ArgumentParser(description="Simple Port Scanner")
        parser.add_argument("ip", help="Target IP address")
        args = parser.parse_args()
        scan_ports(args.ip)

    if __name__ == "__main__":
        main()
```

- **You can now import scan_ports in another tool:**
  - from port_scanner import scan_ports
  - scan_ports("192.168.1.1")

API Interaction

API Interaction in Python involves sending requests to external services (usually web servers) and processing their responses — often to fetch data or perform actions. In cybersecurity, it's especially useful for automating tools, querying threat intel databases, or working with services like VirusTotal or Shodan.

- **Using requests.get() and requests.post():** Python's requests library is a user-friendly HTTP client for sending API requests.
  - **Installation:** pip install requests
  - **GET request (fetch data)**

    -
      ```
      import requests

      url = "https://jsonplaceholder.typicode.com/posts/1"
      response = requests.get(url)

      print(response.status_code)  # 200 means success
      print(response.text)         # Raw response
      ```
  - **POST request (send data)**

    -
      ```
      url = "https://jsonplaceholder.typicode.com/posts"
      data = {"title": "CyberSec", "body": "Testing API", "userId": 1}

      response = requests.post(url, json=data)

      print(response.status_code)
      print(response.json())  # Parses JSON response directly
      ```

- **Parsing API Responses (JSON):** Most modern APIs return **JSON**. Use .json() to parse the response.

```
response = requests.get("https://jsonplaceholder.typicode.com/users/1")
data = response.json()

print(data["name"])        # Get the user's name
print(data["address"]["city"])  # Nested field
```

  - **Looping through a JSON list:**

    -
      ```
      response =
      requests.get("https://jsonplaceholder.typicode.com/posts")
      posts = response.json()

      for post in posts[:3]:
         print(f"ID: {post['id']} | Title: {post['title']}")
      ```
- **Sending Headers and Authentication Tokens:** APIs often require headers, especially for **auth**.
  - **Custom headers**

    -
      ```
      headers = {
         "User-Agent": "CyberSecBot/1.0"
      }
      response = requests.get("https://httpbin.org/headers",
      headers=headers)
      print(response.json())
      ```
  - **Bearer Token Authentication:** Useful for APIs like VirusTotal, Shodan, etc.

    -
      ```
      headers = {
         "Authorization": "Bearer YOUR_API_TOKEN"
      }

      response = requests.get("https://api.example.com/data",
      headers=headers)
      print(response.status_code)
      ```
- **Summary**

○ Send GET request:	requests.get(url)
○ Send POST request:	requests.post(url, ...)
○ Read JSON response:	response.json()
○ Add headers (like tokens):	headers={"key": "val"}

Command-Line and System Interaction

Command-Line and System Interaction in Python refers to using Python to:

- Accept and handle arguments passed from the terminal (command-line).
- Run shell or terminal commands (like ls, ping, or nmap) from Python.
- Capture and process the output of these commands.

This is **very important** in cybersecurity and hacking tools — since many are CLI-based, automating them or interacting with their outputs programmatically can give you a huge advantage.

1. **Using *sys.argv* for Command-Line Arguments :** This allows your script to take inputs directly from the command line.

```
# save as cli_args.py
import sys

print("Script name:", sys.argv[0])  # always the script name
print("Arguments passed:", sys.argv[1:])  # arguments start from index 1
```

- **Run it like this :**
  - python cli_args.py arg1 arg2

```
#Output:
Script name: cli_args.py
Arguments passed: ['arg1', 'arg2']
```

- **Use case (e.g., IP scanner):**

```
import sys
import os

if len(sys.argv) < 2:
    print("Usage: python scanner.py <target_ip>")
    sys.exit(1)

target_ip = sys.argv[1]
os.system(f"ping -c 4 {target_ip}")
```

2. **Running Shell Commands with subprocess**:
   - The subprocess module lets you run shell commands in a safe and controlled way (better than os.system).
     - import subprocess

```
# This runs the command
subprocess.run(["ls", "-l"])
```

- **With a single string (requires shell=True):** Be careful with shell=True if you use untrusted input (to avoid shell injection attacks).
  - subprocess.run("ls -l", shell=True)

3. **Capturing Output from Terminal Tools**
   - If you want to read the output of a command (e.g., nmap, ifconfig, netstat, etc.), use subprocess.check_output() or subprocess.run(..., capture_output=True):
   - import subprocess

```
# Capture command output as text
output = subprocess.check_output(["whoami"], text=True)
print("Current user is:", output.strip())
```

   - Or using subprocess.run:
     - result = subprocess.run(["ifconfig"], capture_output=True, text=True)
       print(result.stdout) # output

N.B: You can also use this to **parse** command output in tools like: netstat, ip a, ping, tcpdump, nmap, etc.

- **Quick Example: Parse Ping Result**
  - import subprocess

```
import subprocess
import sys

if len(sys.argv) != 2:
   print("Usage: python ping_parse.py <ip>")
   sys.exit(1)

ip = sys.argv[1]
result = subprocess.run(["ping", "-c", "2", ip], capture_output=True,
text=True)

if "0 received" in result.stdout:
   print(f"{ip} is DOWN")
else:
   print(f"{ip} is UP")
```