# Anatomy of Python

- **Python General purpose high level language.**
- **paradigm:**
    - procedural.
    - functional.
    - object-oriented programming.
- **Python is high level.**
    - low level languages - high level languages.
        - low level languages: self managed memory.
        - high level languages: language managed memory.
- **Python is interpreted. (some compilation)**
    - compiled Language :
        - compilation -> machine code.
        - faster execution.
        - very few runtime exceptions.
    - interpreted Language :
        - line by line execution
        - slower execution.
        - a lot of runtime exceptions.
- **Python is dynamically typed.**
    - statically typed:

      ```
      int x=5;
      x="mina"; //Error
      ```

    - dynamically typed:

      ```
      x = 5
      x= "mina" # Valid Not Error
      ```

- **Python is strongly typed.**
    - weakly typed:

      ```
      var x="my string"
      var y=5
      console.log(x+y)
      //js Output : my string5
      ```

    - strongly typed:

```
    x = "my string"
    y = 5
    print(x + y) # Python : Error  can't string+int [ don't have Automatic casting ]
```

- **case sensitive**

```
    x=5
    X=10
```

- **Object oriented based (verything in python is an object).**
- **Syntax and Structure**
  - Indentation: Python uses indentation (whitespace) to define code blocks instead of braces {}.
  - Statements: Each line of code is typically a statement. Statements can be simple (e.g., x = 5) or compound (e.g., loops, conditionals).
  - Comments: Use # for single-line comments and """ or ''' for multi-line comments.
- **Memory Management**
  - Python handles memory management automatically using:
    - Garbage Collection: Reclaims unused memory.
    - Reference Counting: Tracks object references.
- **python files : generally ends with .py**
- **you can execute the file using the python cli command**

```
Example:
import gc
gc.collect()  # Manually trigger garbage collection
```

- **Standard**
  - PEP rules: Python enhancement proposals.
  - Default implementation for the standard: CPython.
  - Other Implementations :-
    - IronPython: C#
    - Jython: Java
    - Cython: python compiled into C. C-types
    - PyPy: python (RPython)

## Comparison Table of Python Implementations

- Python has multiple implementations, and they can be broadly classified into interpreters and compilers. Each implementation has its own characteristics, performance optimizations, and use cases.

#### CPython (Standard Python Interpreter)

- Type: Interpreter (with some compilation to bytecode)
- Language: Written in C
- Execution:
- Converts Python code (.py) into bytecode (.pyc).
- The Python Virtual Machine (PVM) executes the bytecode.
- Features:
  - Most widely used and official implementation of Python.
  - Supports C extensions (.so/.dll files).
  - Uses Global Interpreter Lock (GIL), making multi-threading less efficient for CPU-bound tasks. Usage: python my_script.py

| Pros: | Cons: |
|---|---|
| Most compatible with Python libraries. | Slower than compiled languages (C, Java). |
| Stable and well-supported. | GIL prevents full CPU parallelism. |

#### PyPy (JIT Compiler for Faster Execution)

- Type: JIT (Just-In-Time) Compiler
- Language: Written in Python (RPython)
- Execution:
  - Uses Just-In-Time (JIT) compilation, converting Python code into machine code at runtime.
- Features:
  - Much faster than CPython for long-running programs.
  - Optimized memory management.
  - Still supports most Python features.
- Usage: pypy my_script.py

| Pros: | Cons: |
|---|---|
| 2-10x faster execution than CPython for certain workloads. | Not fully compatible with all CPython extensions. |
| Optimized for performance. | Larger memory usage than CPython. |

#### Jython (Python on the Java Virtual Machine)

- Type: Python Compiler to Java Bytecode
- Language: Written in Java
- Execution:
  - Compiles Python code to Java bytecode.
  - Runs on the JVM (Java Virtual Machine).
- Features:
  - Allows seamless integration with Java libraries.
  - No GIL, so it supports true multithreading.

- Usage: jython my_script.py

| Pros: | Cons: |
|---|---|
| Can call Java classes and use Java libraries. | Does not support C extensions (e.g., NumPy, SciPy). |
| Runs in JVM environments. | Slower than CPython for some tasks. |

#### IronPython (Python on .NET)

- Type: Python Compiler to .NET Bytecode
- Language: Written in C#
- Execution:
  - Compiles Python code to .NET Common Intermediate Language (CIL).
  - Runs on the .NET runtime (CLR).
- Features:
  - Allows direct access to .NET libraries (C#, VB.NET).
  - No GIL, so better multithreading.
- Usage: ipy my_script.py

| Pros: | Cons: |
|---|---|
| Fully integrates with the .NET ecosystem. | No C extension support (like Jython). |
| Supports true parallel execution. | Less popular and less actively maintained. |

#### MicroPython (Python for Embedded Systems)

- Type: Lightweight Python Interpreter
- Language: Written in C
- Execution:
  - Designed for microcontrollers (ESP32, Raspberry Pi Pico).
- Features:
  - Optimized for low memory usage.
  - Supports Python 3 but removes some features to save space.
- Usage: micropython my_script.py

| Pros: | Cons: |
|---|---|
| Great for IoT and embedded systems. | Limited library support. |
| Lightweight and fast. | Some standard Python features are missing. |

Comparison Table of Python Implementations

| Implementation | Type | Written In | Speed | GIL? | Best Use Case |
|---|---|---|---|---|---|
| CPython | Interpreter | C | Slow | ✅ Yes | General-purpose (Official) |

| Implementation | Type | Written In | Speed | GIL? | Best Use Case |
|---|---|---|---|---|---|
| PyPy | JIT Compiler | RPython | Fast | ✅ Yes | Performance-intensive apps |
| Jython | Compiler | Java | Medium | ❌ No | Java Integration |
| IronPython | Compiler | C# | Medium | ❌ No | .NET Integration |
| MicroPython | Interpreter | C | Fast | ✅ Yes | Embedded systems |

---

## The Zen of Python, by Tim Peters

```
- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!
```

---

## Python Visualization

### Which One Should You Use?

| Tool | Best For | Visualization Type |
|---|---|---|
| Python Tutor | Step-by-step execution | Stack, heap, variable changes |
| pycallgraph | Function call tracking | Call graph |
| VizTracer | Profiling execution time | Timeline graph |
| pyinstrument | Performance bottlenecks | Hierarchical execution time |

| Tool | Best For | Where to Use |
|---|---|---|
| Python Tutor | Step-by-step visualization with memory usage | Online |

| Tool | Best For | Where to Use |
|---|---|---|
| pdb | Debugging in the terminal | Local scripts |
| breakpoint() | Quick debugging in Python 3.7+ | Local scripts |

## Variable In Python

- variable is a named location in memory used to store data.
- variable is a container to put data into
- Variables can hold different types of data, known as data types.
- Python is dynamically typed, meaning you don't need to explicitly declare the type of a variable; the type is inferred from the value assigned to it.
- A variable is created when you assign a value to it.
- **Variable names must follow specific rules:**
  - Start with a letter (a-z, A-Z) or an underscore (_).
  - Can contain letters, numbers, and underscores.
  - Cannot be a reserved keyword (e.g., if, else, for, etc.).
  - Case-sensitive (e.g., age and Age are different variables).
- **Variable Naming Conventions**
  - Use descriptive names (e.g., user_name instead of un).
  - Use lowercase with underscores for variable names (snake_case).
  - Avoid single-letter names unless they are meaningful (e.g., i for an index).
- Python supports various data types, including numeric, sequence, mapping, set, boolean, and None.
- Use type() to check the data type of a variable.
- Python is dynamically typed, so variables can be reassigned to different data types.
- Use type conversion functions like int(), str(), etc., to convert between data types.
- Python does not have built-in constant variables, but by convention, constants are written in uppercase.

```
    x = 10          # Integer
    name = "Alice"  # String
    is_student = True  # Boolean
#-------------------------------Python is dynamically typed-------------------------
------------#
'''
Python is dynamically typed, so variables
can be reassigned to different data types.
'''
    x = 10        # x is an integer
    x = "Hello"   # x is now a string
    x = 3.14      # x is now a float
#-------------------------------Data Types in Python--------------------------------
-----#
#### Data Types in Python
'''
    data types in python:
        a - simple:
                1 - int:    1
                2 - float:  1.5
                3 - str:    "some string"
                4 - bool:   True, False
                5 - None:   None
        b - containers:
                1 - List:   [1, 2, 3]
                2 - Tuple:  (1, 2)
                3 - Dict:  {"key": "value"} dictionary
                4 - set:   {1, 2}
    to know the type of a variable
    use: type()
'''
# 1. Numeric Types
#    int: Integer numbers (e.g., 10, -5, 1000).
#    float: Floating-point numbers (e.g., 3.14, -0.001, 2.0).
#    complex: Complex numbers (e.g., 1 + 2j).
a = 5          # int
b = 3.14       # float
c = 2 + 3j     # complex
# 2. Sequence Types
#    str: A sequence of characters (e.g., "Hello", 'Python').
#    list: An ordered, mutable collection of items (e.g., [1, 2, 3]).
#    tuple: An ordered, immutable collection of items (e.g., (1, 2, 3)).
name = "Alice"          # str
numbers = [1, 2, 3]     # list
coordinates = (4, 5)    # tuple
# 3. Mapping Type : dict: A collection of key-value pairs (e.g., {"name": "Alice", "age": 25}).
person = {"name": "Alice", "age": 25}  # dict
# 4. Set Types: set: An unordered collection of unique items (e.g., {1, 2, 3}).
unique_numbers = {1, 2, 3}  # set
# 5. Boolean Type : bool: Represents True or False.
    is_student = True  # bool
# 6. None Type:None: Represents the absence of a value or a null value.
result = None  # NoneType
#-------------------------------Check data types-------------------------------------#
# Variables
name = "Alice"          # str
age = 25               # int
height = 5.6           # float
is_student = True      # bool
hobbies = ["reading", "coding"]  # list
address = ("123 Main St", "City")  # tuple
person = {"name": "Alice", "age": 25}  # dict
# Check data types
print(type(name))        # <class 'str'>
print(type(age))         # <class 'int'>
print(type(height))      # <class 'float'>
print(type(is_student))  # <class 'bool'>
print(type(hobbies))     # <class 'list'>
print(type(address))     # <class 'tuple'>
print(type(person))      # <class 'dict'>
#------------------------------Type Conversion (Casting)--------------------------
----------#
##### Type Conversion (Casting)
# Convert string to integer
age = "25"
age_int = int(age)
# Convert integer to string
count = 10
count_str = str(count)
# Convert float to integer
pi = 3.14
pi_int = int(pi)  # Truncates the decimal part

#------------------------------Constants in Python--------------------------------
----#
#Constants in Python
#Python does not have built-in constant variables, but by convention, constants are
written in uppercase.
PI = 3.14159
GRAVITY = 9.8
```

## DataTypes In Python

### Integer

#### Integer Caching Range (-5 to 256)

- Python interns (caches) integers in the range -5 to 256.
- If a number is within this range, Python reuses the same object in memory.
- In CPython, small integers are often cached for optimization purposes. This means that certain small integer values (typically between -5 and 256) are pre-allocated and reused. When you assign the same small integer value to different variables, they may refer to the same object in memory.
- Integers outside the range -5 to 256 are not cached by default.
- A new object is created each time.
- Caching reduces memory usage and improves performance.

```
a=5
b=5
print(f"{a is b=}") # prints True
a=a+1
b=b+1
print(f"{a is b=}") # prints True
a = 256
```

```
b = 256
print(f"{a is b=}") # prints True
a=a+1
b=b+1
print(f"{a is b=}") # prints False
a = 257
b= 257
print(f"{a is b=}") # prints True
a=a+1
b=b+1
print(f"{a is b=}") # prints False
#===================================================#
x=-5
y=-5
while x is y :
    x=x+1
    y=y+1
print(f"{x=} {y=}") #257 #257
#================================================Type Casting
================================
print(int("400"))
print(int("  400_000  ")) # remove spaces, accepts _
#print(int("404s")) # Error
print(int("101", 2)) #Binary Format
print(int("F", 16)) # Hexa Format
```

## Floating

**limitations of: IEEE754:**

**1. float + the double of that float**

- floating point representation full number, floating point number

```
    x=0.1+0.2
    print(x) #0.30000000000000004
    print(round(x,2) == 0.3 ) #True
    print(x == 0.3 ) #False
```

**2. accuracy for floating point numbers more than quadrillion:**

```
    x=0.9999999999999999999999999999999999999999999999999
    print(x) #1.0
```

**Casting**

```
print(float("3.5"))
```

## String

- **Key Notes:**
  - These functions are built-in, meaning you don’t need to import any modules to use them.
  - Strings in Python are immutable, so these functions return new strings rather than modifying the original.
  - For more advanced string manipulation, you can use the re module (regular expressions).

| Function | Description | Example | Output |
|---|---|---|---|
| len() | Returns the length of a string. | len("hello") | 5 |
| str() | Converts a value to a string. | str(42) | "42" |
| ord() | Returns the Unicode code point of a character. | ord("A") | 65 |
| chr() | Returns the character corresponding to a Unicode code point. | chr(65) | "A" |
| upper() | Converts a string to uppercase. | "hello".upper() | "HELLO" |
| lower() | Converts a string to lowercase. | "HELLO".lower() | "hello" |
| capitalize() | Converts the first character to uppercase and the rest to lowercase. | "hello world".capitalize() | "Hello world" |
| title() | Converts the first character of each word to uppercase. | "hello world".title() | "Hello World" |
| strip() | Removes leading and trailing whitespace (or specified characters). | " hello ".strip() | "hello" |
| lstrip() | Removes leading whitespace (or specified characters). | " hello ".lstrip() | "hello " |
| rstrip() | Removes trailing whitespace (or specified characters). | " hello ".rstrip() | " hello" |
| replace() | Replaces occurrences of a substring with another | "hello world".replace("world", "Python") | "hello Python" |

| Function | Description | Example | Output |
|---|---|---|---|
| | substring. | | |
| split() | Splits a string into a list of substrings based on a delimiter. | "apple,banana,cherry".split(",") | ['apple', 'banana', 'cherry'] |
| join() | Joins elements of an iterable into a single string using a specified separator. | ", ".join(["apple", "banana", "cherry"]) | "apple, banana, cherry" |
| find() | Returns the index of the first occurrence of a substring (or -1 if not found). | "hello world".find("world") | 6 |
| index() | Similar to find(), but raises a ValueError if the substring is not found. | "hello world".index("world") | 6 |
| count() | Returns the number of occurrences of a substring in a string. | "hello world".count("l") | 3 |
| startswith() | Checks if a string starts with a specified substring. | "hello world".startswith("hello") | True |
| endswith() | Checks if a string ends with a specified substring. | "hello world".endswith("world") | True |
| isalpha() | Checks if all characters in a string are alphabetic. | "hello".isalpha() | True |
| isdigit() | Checks if all characters in a string are digits. | "12345".isdigit() | True |
| isalnum() | Checks if all characters in a string are alphanumeric (letters or numbers). | "hello123".isalnum() | True |
| isspace() | Checks if all characters in a string are whitespace. | " ".isspace() | True |
| isupper() | Checks if all characters in a string are uppercase. | "HELLO".isupper() | True |
| islower() | Checks if all characters in a string are lowercase. | "hello".islower() | True |

| Function | Description | Example | Output |
|---|---|---|---|
| zfill() | Pads a string with zeros on the left until it reaches a specified length. | "42".zfill(5) | "00042" |
| format() | Formats a string using placeholders. | "My name is {} and I am {} years old.".format("Alice", 30) | "My name is Alice and I am 30 years old." |
| f-strings | Modern way to format strings (Python 3.6+). | name = "Alice"; age = 30; f"My name is {name} and I am {age} years old." | "My name is Alice and I am 30 years old." |
| encode() | Encodes a string into bytes using a specified encoding (e.g., UTF-8). | "hello".encode("utf-8") | b'hello' |
| decode() | Decodes bytes into a string using a specified encoding. | b'hello'.decode("utf-8") | "hello" |

```
#=======================================indexing====================================
==========#
text = "Python"
print(text[0])  # Output: 'P'
print(text[-1]) # Output: 'n' (negative indexing)
#=======================================Multi Line
Comment==============================================#
''' '''        or  """ """"
#=======================================Slicing
==============================================#
text = "Python"
print(text[1:4])  # Output: 'yth' (from index 1 to 3)
print(text[:3])   # Output: 'Pyt' (from start to index 2)
print(text[3:])   # Output: 'hon' (from index 3 to end)
print(text[::2])  # Output: 'Pto' (step of 2)
#=======================================Concatenation===============================
===============#
# You can combine strings using the + operator.
s1 = "Hello"
s2 = "World"
result = s1 + " " + s2
print(result)  # Output: "Hello World"
#=======================================Repetition==================================
============#
#You can repeat a string using the * operator.
text = "Python"
print(text * 3)  # Output: "PythonPythonPython
```

```
#======================================String
Membership=============================================#
#Check if a substring exists in a string using the in keyword.
text = "Python"
print("th" in text)  # Output: True
print("z" in text)   # Output: False

# in to check on the presence of the stri
x="eslamreda.info"
if 'e' in x
    print("Found ")

#======================================Raw
Strings=============================================#
# Use raw strings to treat backslashes as literal characters.
print(r"C:\Users\Name")  # Output: C:\Users\Name

#======================================String
Immutability=============================================#
#Strings are immutable, meaning you cannot change them in place. Instead, you create
new strings.
c = "hello world!"
d = "hello world!"
# c[0] = 'H'  # This will raise an error
print(f"{c==d=} {id(c) == id(d)=}")  #c==d=True id(c) == id(d)=True
c='H'+c[1:]
d='H'+d[1:]
print(f"{c==d=} {id(c) == id(d)=}")  #c==d=True id(c) == id(d)=False
```

### List

- built-in list functions Key Notes:
  - **In-place operations:** Functions like append(), extend(), insert(), remove(), pop(), clear(), sort(), and reverse() modify the original list.
  - **Non-in-place operations:** Functions like sorted(), copy(), list(), map(), filter(), and reversed() return a new list or object without modifying the original list.

| Function/Method | Description | Example | Output |
|---|---|---|---|
| len() | Returns the number of elements in a list. | len([1, 2, 3]) | 3 |
| append() | Adds an element to the end of the list. | my_list = [1, 2]; my_list.append(3) | [1, 2, 3] |

| Function/Method | Description | Example | Output |
|---|---|---|---|
| extend() | Adds all elements of an iterable to the end of the list. | my_list = [1, 2]; my_list.extend([3, 4]) | [1, 2, 3, 4] |
| insert() | Inserts an element at a specific index. | my_list = [1, 2]; my_list.insert(1, 1.5) | [1, 1.5, 2] |
| remove() | Removes the first occurrence of a specific element. | my_list = [1, 2, 3]; my_list.remove(2) | [1, 3] |
| pop() | Removes and returns the element at a specific index (default is the last element). | my_list = [1, 2, 3]; my_list.pop(1) | 2 (list becomes [1, 3]) |
| clear() | Removes all elements from the list. | my_list = [1, 2, 3]; my_list.clear() | [] |
| index() | Returns the index of the first occurrence of a specific element. | my_list = [1, 2, 3]; my_list.index(2) | 1 |
| count() | Returns the number of occurrences of a specific element. | my_list = [1, 2, 2, 3]; my_list.count(2) | 2 |
| sort() | Sorts the list in ascending order (in-place). | my_list = [3, 1, 2]; my_list.sort() | [1, 2, 3] |
| reverse() | Reverses the order of elements in the list (in-place). | my_list = [1, 2, 3]; my_list.reverse() | [3, 2, 1] |
| copy() | Returns a shallow copy of the list. | my_list = [1, 2, 3]; new_list = my_list.copy() | [1, 2, 3] |
| list() | Converts an iterable (e.g., tuple, string) into a list. | list("hello") | ['h', 'e', 'l', 'l', 'o'] |
| sum() | Returns the sum of all elements in the list. | sum([1, 2, 3]) | 6 |
| min() | Returns the smallest element in the list. | min([1, 2, 3]) | 1 |
| max() | Returns the largest element in the list. | max([1, 2, 3]) | 3 |
| sorted() | Returns a new sorted list (does not modify the original list). | sorted([3, 1, 2]) | [1, 2, 3] |
| any() | Returns True if at least one element in the list is true. | any([0, False, 1]) | True |
| all() | Returns True if all elements in the list are true. | all([1, True, 0]) | False |

| Function/Method | Description | Example | Output |
|---|---|---|---|
| enumerate() | Returns an enumerate object (index, value pairs). | list(enumerate(['a', 'b', 'c'])) | [(0, 'a'), (1, 'b'), (2, 'c')] |
| filter() | Filters elements based on a condition. | list(filter(lambda x: x > 1, [1, 2, 3])) | [2, 3] |
| map() | Applies a function to all elements in the list. | list(map(lambda x: x * 2, [1, 2, 3])) | [2, 4, 6] |
| zip() | Combines multiple iterables into tuples. | list(zip([1, 2], ['a', 'b'])) | [(1, 'a'), (2, 'b')] |
| slice() | Returns a slice object for slicing lists. | my_list = [1, 2, 3, 4]; my_list[slice(1, 3)] | [2, 3] |
| reversed() | Returns a reverse iterator for the list. | list(reversed([1, 2, 3])) | True |

```
#====================================== Create List
======================================#
a = [1, 2, 3, 4, 5]# List of integers
b = ['apple', 'banana', 'cherry']# List of strings
c = [1, 'hello', 3.14, True]# Mixed data types
a = [2] * 5 # Create a list [2, 2, 2, 2, 2]
b = [0] * 7 # Create a list [0, 0, 0, 0, 0, 0, 0]
a = list((1, 2, 3, 'apple', 4.5))  # Create List From a tuple
#=====================================List to
String========================================#
my_list = ["Python", "is", "fun"]
result = " ".join(my_list)
print(result)  # Output: "Python is fun"
#====================================== Indexing List
======================================#
a = [10, 20, 30, 40, 50]
print(a[0])  # Access first element
print(a[2])  # Access Third element
print(a[-1])# Access last element
#====================================== Adding Element in  List
======================================#
a = []# Initialize an empty list
a.append(10)  # Adding 10 to end of list
print("After append(10):", a)  #[10]
a.insert(0, 5)# Inserting 5 at index 0
print("After insert(0, 5):", a)#[5,10]
a.append([15, 20, 25])  # Adding multiple elements  [15, 20, 25] at the end
print("After extend([15, 20, 25]):", a) #[5, 10, [15, 20, 25]]
a.extend([15, 20, 25])  # Adding multiple elements  [15, 20, 25] at the end
print("After extend([15, 20, 25]):", a) #[5, 10, [15, 20, 25], 15, 20, 25
#======================================Updating Elements into
List======================================#
a = [10, 20, 30, 40, 50]
a[1] = 25 # Change the second element
print(a)  #[10, 25, 30, 40, 50]
#======================================Removing Elements from
List======================================#
# remove(): Removes the first occurrence of an element.
# pop(): Removes the element at a specific index or the last element if no index is
specified.
# del statement: Deletes an element at a specified index.
# clear() - Removes all elements
a = [10, 20, 30, 40, 50]
a.remove(30) # Removes the first occurrence of 30
print("After remove(30):", a)#[10, 20, 40, 50]

popped_val = a.pop(1)  # Removes the element at index 1 (20)
print("Popped element:", popped_val) #20
print("After pop(1):", a) #[10, 40, 50]

del a[0]  # Deletes the first element (10)
print("After del a[0]:", a) #[40, 50]
a.clear()
print("After clear() a : " ,a) #[]
#==========================================Concatenation & Repetition
======================================
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2
print(result)  # Output: [1, 2, 3, 4, 5, 6]

my_list = [1, 2, 3]
print(my_list * 2)  # Output: [1, 2, 3, 1, 2, 3]
#======================================Checking Membership (in and not
in)===================#
my_list = [10, 20, 30]
print(20 in my_list)  # Output: True
print(40 not in my_list)  # Output: True

#======================================Sorting and Reversing a
List===========================
#a) sort() - Sorts the list in ascending order
#b) sort(reverse=True) - Sorts in descending order
#c) sorted() - Returns a new sorted list (original remains unchanged)
#d) reverse() - Reverses the list
nums = [3, 1, 4, 1, 5, 9]
nums.sort()
print(nums)  # Output: [1, 1, 3, 4, 5, 9]
nums.sort(reverse=True)
print(nums)  # Output: [9, 5, 4, 3, 1, 1]
#====================================================================#
str_list = [ "Mohamed", "Abdallah", "Abdallah", "Mostafa","Ahmed"]
```

```
str_list.sort(key=len)
print(str_list)
#=====================================================================#
nums = [3, 1, 4, 1, 5, 9]
sorted_nums = sorted(nums)
print(sorted_nums)  # Output: [1, 1, 3, 4, 5, 9]
print(nums)  # Original list remains unchanged
#=====================================================================#
nums.reverse()
print(nums)  # Output: [9, 5, 4, 3, 1, 1]
#==========================================Copying a List
=============================================
l=[1,2,3,5]
lcopy=l #Shallow Copy
print(id(l)==id(lcopy)) #True

lcopy=l[::] #deep copy
print("l[::] : " , id(l)==id(lcopy)) #False

lcopy=l.copy() #deep copy
print("l.copy() : " , id(l)==id(lcopy)) #False
import copy
lcopy=copy.copy(l) #deep copy
print("copy.copy(l) : " , id(l)==id(lcopy)) #False
#Copy Nested List
l=[[1,2,3,5],1,2,3,5]
lcopy=l[::] # or lcopy=l.copy() #or lcopy=copy.copy(l)
print("List : ",id(l)==id(lcopy)) #False
print("Nested List : " ,id(l[0])==id(lcopy[0])) #True

lcopy=l[::]
lcopy=copy.deepcopy(l) #Full deep Copy
print("List : ",id(l)==id(lcopy)) #False
print("Nested List : " ,id(l[0])==id(lcopy[0])) #False

#==========================================Finding Maximum, Minimum, and Sum
=========================
numbers = [10, 20, 30, 40, 50]
print(max(numbers))  # Output: 50
print(min(numbers))  # Output: 10
print(sum(numbers))  # Output: 150
#==========================================  Counting and Finding
Index=========================
my_list = [10, 20, 30, 20, 40]
# Count occurrences of 20
print(my_list.count(20))  # Output: 2
# Find index of first occurrence of 30
print(my_list.index(30))  # Output: 2
#==========================================  Destruct List
========================================#
my_list = [1, 2]
```

```
a, b = my_list
print(f"{a=} {b=}") #a=1 b=2

#Swap
a = 5
b = 6
b, a = [a, b]
print(f"{a=} {b=}") #a=6 b=5
#a, b= [1, 2, 3] # ========> Error
a, b, _ = [1, 2, 3]
print(f"{a=} {b=}") #a=6 b=2
a, *_, b = [1, 2, 3, 4, 5, 6, 7]
print(f"{a=} {b=}") #a=1 b=7
a, *c, b = [1, 2, 3, 4, 5, 6, 7]
print(f"{a=} {c=} {b=}") #a=1 c=[2, 3, 4, 5, 6] b=7

#======================================Nested
List======================================#
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[1][2])#6 # Access element at row 2, column 3

#======================================List Comprehension (Compact List
Creation)===========================
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

## Tuple

- tuples are Immutable , comparable and Hashable
- Why?
    - Because it's immutable
    - less size than list

```


t=('a','b')
print(type(t))# <class 'tuple'>

t= tuple('name')
print(t) # ('n','a','m','e')

# Tuple of One Element
t=('a')
print(type(t))# <class 'str'>
t=('a',)
```

```
t='a',
print(type(t))# <class 'tuple'>

# Acess by index
t=('a','b')
print(t[0]) #a
#Immutable
#t[0]='N' #Error

t2= ('N',)+t[1:]

#Compared
print((0,1)==(0,1))#True
print((0,1)==(0,2))#False
print((0,1)<(0,2))#True
print((0,1,2,3)<(0,2,1,1)) #True

#====================================using Tuple in Assigment======================
a = 5
b = 6
b, a = a, b
print(f"{a=} {b=}") #a=6 b=5
#a, b= 1, 2, 3 # ========> Error
a, b, _ = 1, 2, 3
print(f"{a=} {b=}") #a=6 b=2
a, *_, b = 1, 2, 3, 4, 5, 6, 7
print(f"{a=} {b=}") #a=1 b=7
a, *c, b = 1, 2, 3, 4, 5, 6, 7
print(f"{a=} {c=} {b=}") #a=1 c=[2, 3, 4, 5, 6] b=7
```

## Dictionary

- **Key Notes:**
  - Views: Methods like keys(), values(), and items() return view objects, which are dynamic and reflect changes to the dictionary.
  - Dictionary keys are case sensitive: the same name but different cases of Key will be treated distinctly.
  - Keys must be immutable: This means keys can be strings, numbers, or tuples but not lists.
  - Keys must be unique: Duplicate keys are not allowed and any duplicate key will overwrite the previous value.
  - Order: As of Python 3.7+, dictionaries maintain insertion order.
  - Dictionary internally uses Hashing. Hence, operations like search, insert, delete can be

| Function/Method | Description | Example | Output |
|---|---|---|---|
| len() | Returns the number of key-value pairs in the dictionary. | len({"a": 1, "b": 2}) | 2 |

| Function/Method | Description | Example | Output |
|---|---|---|---|
| dict() | Creates a dictionary from an iterable or keyword arguments. | dict(a=1, b=2) | {'a': 1, 'b': 2} |
| keys() | Returns a view of all keys in the dictionary. | {"a": 1, "b": 2}.keys() | dict_keys(['a', 'b']) |
| values() | Returns a view of all values in the dictionary. | {"a": 1, "b": 2}.values() | dict_values([1, 2]) |
| items() | Returns a view of all key-value pairs as tuples. | {"a": 1, "b": 2}.items() | dict_items([('a', 1), ('b', 2)]) |
| get() | Returns the value for a key. If the key doesn't exist, returns a default value. | {"a": 1, "b": 2}.get("a") {"a": 1, "b": 2}.get("c", 0) | 1 0 |
| setdefault() | Returns the value for a key. If the key doesn't exist, inserts it with a default value. | d = {"a": 1}; d.setdefault("b", 2) | 2 (d becomes {'a': 1, 'b': 2}) |
| update() | Updates the dictionary with key-value pairs from another dictionary or iterable. | d = {"a": 1}; d.update({"b": 2}) | {'a': 1, 'b': 2} |
| pop() | Removes and returns the value for a key. Raises KeyError if the key doesn't exist. | d = {"a": 1, "b": 2}; d.pop("a") | 1 (d becomes {'b': 2}) |
| popitem() | Removes and returns the last inserted key-value pair as a tuple. | d = {"a": 1, "b": 2}; d.popitem() | ('b', 2) (d becomes {'a': 1}) |
| clear() | Removes all key-value pairs from the dictionary. | d = {"a": 1, "b": 2}; d.clear() | {} |
| copy() | Returns a shallow copy of the dictionary. | d = {"a": 1}; new_d = d.copy() | {'a': 1} |
| fromkeys() | Creates a new dictionary with keys from an iterable and a default value. | dict.fromkeys(["a", "b"], 0) | {'a': 0, 'b': 0} |
| in | Checks if a key exists in the dictionary. | "a" in {"a": 1, "b": 2} | True |
| not in | Checks if a key does not exist in the dictionary. | "c" not in {"a": 1, "b": 2} | True |
| del | Deletes a key-value pair from the dictionary. | d = {"a": 1, "b": 2}; del d["a"] | {'b': 2} |

| Function/Method | Description | Example | Output |
|---|---|---|---|
| sorted() | Returns a sorted list of keys (or items) in the dictionary. | sorted({"b": 2, "a": 1})<br>sorted({"b": 2, "a": 1}.items()) | ['a', 'b'] [('a', 1), ('b', 2)] |
| any() | Returns True if any key in the dictionary is true. | any({0: False, 1: True}) | True |
| all() | Returns True if all keys in the dictionary are true. | all({1: True, 2: True}) | True |
| max() | Returns the maximum key in the dictionary. | max({"a": 1, "b": 2}) | 'b' |
| min() | Returns the minimum key in the dictionary. | min({"a": 1, "b": 2}) | 'a' |
| sum() | Returns the sum of all keys (if numeric) in the dictionary. | sum({1: "a", 2: "b"}) | 3 |
| zip() | Combines keys and values into tuples. | dict(zip(["a", "b"], [1, 2])) | {'a': 1, 'b': 2} |

```
#Dictionary value types
# keys in a dictionary must always be an immutable data type, such as strings,
numbers, or tuples.
# values in a dictionary to be any type
dictionary = {
  1: 'hello',
  'two': True,
  '3': [1, 2, 3],
  'Four': {'fun': 'addition'},
  5.0: 5.5
}
#=============================Acessing=====================
my_dict = {"name": "Alice", "age": 25}
# Accessing an existing key
print(my_dict.get("name"))  # Output: Alice
print(my_dict["name"])  # Output: Alice
# Accessing a non-existent key
print(my_dict.get("address"))  # Output: None (key doesn't exist)
#print(my_dict["address"])  # Output: Error
if 'address' in my_dict: print(my_dict['address'])#Avoid KeyError
# Providing a default value
print(my_dict.get("address", "Unknown"))  # Output: Unknown
#=============================Add and Updates======================
# Use dict['key'] = value when updating or adding a single key-value pair.
# Use dict.update() when you need to update multiple keys at once or want to accept
a dictionary dynamically.
            #============= dict[key]=vlaue ===========#
```

```
person = {"name": "Alice", "age": 25}
person["age"] = 26 # Updating an existing key
person["city"] = "New York" # Adding a new key-value pair
print(person) # {'name': 'Alice', 'age': 26, 'city': 'New York'}
            #=============dict.update({key:vlaue})===========#
person = {"name": "Alice", "age": 25}
person.update({"age": 26, "city": "New York"})# Updating multiple values at once
person.update(country="USA", hobby="reading")# Using keyword arguments
print(person)
# {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA', 'hobby':
'reading'}
#================Merging dictionaries ====================

dict1 = {'color': 'blue', 'shape': 'circle'}
dict2 = {'color': 'red', 'number': 42}
dict1.update(dict2)
print(dict1)
            #===============================#
dict1 = {'color': 'blue', 'shape': 'circle'}
dict2 = {'color': 'red', 'number': 42}
dict1={**dict1,**dict2}
print(dict1)
#{'color': 'red', 'shape': 'circle', 'number': 42}

#==================== Removing Dictionary Items==================
#del: Removes an item by key.
#pop(): Removes an item by key and returns its value.
#clear(): Empties the dictionary.
#popitem(): Removes and returns the last key-value pair.
d = {1: 'Geeks', 2: 'For', 3: 'Geeks', 'age':22}
del d["age"]# Using del to remove an item
print(d) #{1: 'Geeks', 2: 'For', 3: 'Geeks'}
val = d.pop(1)# Using pop() to remove an item and return the value
print(val) #'Geeks'

# Using popitem to removes and returns
key, val = d.popitem()# the last key-value pair.
print(f"Key: {key}, Value: {val}")

d.clear()# Clear all items from the dictionary
print(d)

#=========================Dictionary Key-Value Methods=========#
ex_dict = {"a": "anteater", "b": "bumblebee", "c": "cheetah"}
ex_dict.keys()
# dict_keys(["a","b","c"])
ex_dict.values()
# dict_values(["anteater", "bumblebee", "cheetah"])
ex_dict.items()
# dict_items([("a","anteater"),("b","bumblebee"),("c","cheetah")]
```

## Set

- Sets are unordered: Elements in a set do not have a specific order.
- Sets contain unique elements: Duplicates are automatically removed.
- Sets are mutable: Except for frozenset, which is immutable.
- Set operations: Sets support mathematical operations like union, intersection, and difference.

| Function/Method | Description | Example | Output |
|---|---|---|---|
| len() | Returns the number of elements in the set. | len({1, 2, 3}) | 3 |
| set() | Creates a set from an iterable (e.g., list, tuple, string). | set([1, 2, 2, 3]) | {1, 2, 3} |
| add() | Adds a single element to the set. | s = {1, 2}; s.add(3) | {1, 2, 3} |
| update() | Adds multiple elements from an iterable to the set. | s = {1, 2}; s.update([3, 4]) | {1, 2, 3, 4} |
| remove() | Removes a specific element from the set. Raises KeyError if the element doesn't exist. | s = {1, 2, 3}; s.remove(2) | {1, 3} |
| discard() | Removes a specific element from the set (does not raise an error if the element doesn't exist). | s = {1, 2, 3}; s.discard(4) | {1, 2, 3} |
| pop() | Removes and returns an arbitrary element from the set. Raises KeyError if the set is empty. | s = {1, 2, 3}; s.pop() | 1 (set becomes {2, 3}) |
| clear() | Removes all elements from the set. | s = {1, 2, 3}; s.clear() | set() |
| copy() | Returns a shallow copy of the set. | s = {1, 2, 3}; new_s = s.copy() | {1, 2, 3} |
| union() | Returns a new set containing all elements from both sets. | {1, 2}.union({2, 3}) | {1, 2, 3} |
| intersection() | Returns a new set containing common elements between two sets. | {1, 2}.intersection({2, 3}) | {2} |
| difference() | Returns a new set containing elements in the first set but not in the second. | {1, 2}.difference({2, 3}) | {1} |

| Function/Method | Description | Example | Output |
|---|---|---|---|
| symmetric_difference() | Returns a new set containing elements in either set but not in both. | {1, 2}.symmetric_difference({2, 3}) | {1, 3} |
| issubset() | Checks if all elements of the set are present in another set. | {1, 2}.issubset({1, 2, 3}) | True |
| issuperset() | Checks if the set contains all elements of another set. | {1, 2, 3}.issuperset({1, 2}) | True |
| isdisjoint() | Checks if two sets have no common elements. | {1, 2}.isdisjoint({3, 4}) | True |
| in | Checks if an element exists in the set. | 2 in {1, 2, 3} | True |
| not in | Checks if an element does not exist in the set. | 4 not in {1, 2, 3} | True |
| frozenset() | Creates an immutable set. | f = frozenset([1, 2, 3]) | frozenset({1, 2, 3}) |
| sorted() | Returns a sorted list of elements in the set. | sorted({3, 1, 2}) | [1, 2, 3] |
| any() | Returns True if at least one element in the set is true. | any({0, False, 1}) | True |
| all() | Returns True if all elements in the set are true. | all({1, True, 0}) | False |
| max() | Returns the maximum element in the set. | max({1, 2, 3}) | 3 |
| min() | Returns the minimum element in the set. | min({1, 2, 3}) | 1 |
| sum() | Returns the sum of all elements in the set. | sum({1, 2, 3}) | 6 |

```
#=============================Empty Set ===============================
set_data={}
print(type(set_data)) #<class dict>
set_data=set()
print(type(set_data)) #<class set>

s = set(["a", "b", "c"])
print(s)
# Adding element to the set
```

```
s.add("d")
print(s)

s = {"Geeks", "for", "Geeks"}
print(s)# a set cannot have duplicate values

# values of a set cannot be changed
s[1] = "Hello"
print(s)
#=============================Operation on Set ===============================
set1={1,2,3,5}
set2={3,4,5}
print(set1|set2) # Union : {1, 2, 3, 4, 5}
print(set1.intersection(set2))# Intersection : {3, 5}
print(set1&set2) # Intersection : {3, 5}

print(set1^set2) # Symmetric Difference : {1, 2, 4}
print(set1.difference(set2))# Difference : {1, 2}
print(set1-set2) # Difference : {1, 2}
print(set2.difference(set1)) # Difference : {4}
print(set2-set1) # Difference : {4}
# my_set={1,5,3,2}
# print(my_set.intersection({1, 2}))
# print(my_set.union({1, 2, 3}))

#============================check for duplicates========================
my_list = [1, 2, 3, 4, 5]
print(len(my_list) == len(set(my_list)))
#===========================Remove Duplications =========================
# Input string with duplicates
input_string = "hello world"
# Remove duplicates using a set
unique_chars = set(input_string)
# Convert the set back to a string
result_string = "".join(unique_chars)
print(result_string)
```

## bool in python

**1.Truthiness and Falsiness:**

- In Python, certain values are considered "falsy" (evaluate to False), while others are "truthy" (evaluate to True).

**Falsy Values:**

False , None , 0 (integer) , 0.0 (float) , "" (empty string) , [] (empty list) , () (empty tuple) , {} (empty dictionary) , set() (empty set)

#### Truthy Values:

Everything else is considered truthy.

### 2.Boolean Operations:

- Python supports logical operations with Boolean values: and: Returns True if both operands are True. or: Returns True if at least one operand is True. not: Negates the Boolean value.

### 3.Comparison Operators:

- Comparison operators return Boolean values (True or False): == (equal to) != (not equal to)

> (greater than) < (less than) = (greater than or equal to) <= (less than or equal to)

### 4.Short Circuit

- and: Stops at the first False value and returns it. If all values are True, returns the last value.
- or: Stops at the first True value and returns it. If all values are False, returns the last value.

#### 4.1. Short-circuiting is useful for:

- Improving performance by avoiding unnecessary computations.
- Preventing errors (e.g., division by zero).
- Writing concise and efficient code.

#### 4.2. Returning Non-Boolean Values

```
# `and` returns the first falsy value or the last value
print(0 and 10)  # 0 (first falsy value)
print(1 and 2 and 3)  # 3 (all truthy, returns last value)
# `or` returns the first truthy value or the last value
print(0 or 10)  # 10 (first truthy value)
print(0 or False or None)  # None (all falsy, returns last value)
```

### 4.3. Avoiding Errors with Short-Circuit

```
x = 0
y = 10
# Without short-circuit (would raise an error)
# result = (y / x > 2)  # ZeroDivisionError

# With short-circuit
```

```
result = x != 0 and (y / x > 2)  # Safe because x != 0 is False
print(result)  # False
```

## Output VS Input in Python

- input and output (I/O) refer to how a program interacts with the user or external data sources.

| Aspect | Input | Output |
|---|---|---|
| Purpose | Provides data to the program. | Displays or saves data from the program. |
| Common Methods | input(), file reading, APIs, etc. | print(), file writing, logging, etc. |
| Data Flow | Into the program. | Out of the program. |
| Examples | User input, file input, API requests. | Console output, file output, logs. |

### Common Ways to Get Input:

- Input refers to the data that is provided to the program, typically by the user or from an external source (e.g., a file, database, or API).

#### 1.Input in Python

- The input() function is used to get user input.
- The data is always received as a string unless explicitly converted.
-

#### How to Change the Type of Input in Python

```
    # Taking input as int
    # Typecasting to int
    n = int(input("How many roses?: "))
    print(n)
    # Taking input as float
    # Typecasting to float
    price = float(input("Price of each rose?: "))
    print(price)
```

#### Take Multiple Input in Python

```
    # taking two inputs at a time
    x, y = input("Enter two values: ").split()
    print("Number of boys: ", x)
    print("Number of girls: ", y)

    # taking three inputs at a time
    x, y, z = input("Enter three values: ").split()
    print("Total number of students: ", x)
    print("Number of boys is : ", y)
    print("Number of girls is : ", z)
```

**2.Command-Line Arguments:**

- Use the sys.argv list to access command-line arguments passed to the script.

```
import sys
print("Script name:", sys.argv[0])
print("Arguments:", sys.argv[1:])
```

**3.Reading from Files:**

- You can read data from files using methods like open() and read().

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

**4.APIs or External Data Sources:**

- Libraries like requests can be used to fetch data from APIs or web services.

```
    import requests  #pip install requests
    response = requests.get("https://api.example.com/data")
    print(response.json())
```

### Common Ways to Get Ouput:

**1. Printing to the Console: Using print() Function:**

- The print() function is the most common way to display output to the console.

```
print("Hello, World!")
print("The value is:", 42)
```

- Using sep and end parameter

```
# end Parameter with '@'
print("Python", end='@')
print("GeeksforGeeks")
# Seprating with Comma
print('G', 'F', 'G', sep='')
# for formatting a date
print('09', '12', '2016', sep='-')
# another example
print('pratik', 'geeksforgeeks', sep='@')
```

**TypeError when combining numbers and strings:**

- TypeError: can only concatenate str (not "int") to str print( "I am " + 25 + " years old.")

**Why Does This Happen?**

- The + operator behaves differently depending on the data types: For strings, it performs concatenation. For numbers, it performs addition.
- Python does not automatically convert numbers to strings when using +.

**How fix it ?**

To avoid the TypeError when combining numbers and strings:

- Use str() to convert numbers to strings.
- Use f-strings, format(), or % for cleaner and more readable code.
- Use commas in print() for quick output.

```
#Use f-strings, format(), or % formatting to create formatted output.
    f_name="mina"
    l_name="nagy"
    age=30
    # print("Hello "+f_name+" "+l_name+" age= "+age) #Error
    #Solution 1 : Convert to String using str() and +
        print("Hello "+f_name+" "+l_name+" age= "+str(age) ) #Hello mina nagy age=
30
    #Solution 2 : Use Commas in print()
         print("Hello ",f_name," ",l_name," age=",age) #Hello  mina   nagy  age= 30
    #Solution 3 : Use the % Operator (Older Style)
        print("Hello %s %s age= %d "%(f_name,l_name,age)) #Hello mina nagy age= 30
    #Solution 4 : String Formate [using format()]
        print("{} {} is {} years old.".format(f_name,l_name, age))  # mina nagy is
30 years old.
        print("Hello {1} {0} age= {2} ".format(f_name,l_name,age)) #Hello nagy mina
```

```
age= 30
        print("Hello {f_name} {l_name} age= {age}
".format(f_name=f_name,l_name=l_name,age=age)) # mina nagy is 30 years old.
    #Solution 4 : Use f-strings (Formatted String Literals) : V3.6
        print(f"Hello {f_name} {l_name} age= {age} ") # mina nagy is 30 years old.
```

**2. Writing to Files:**

- Use the open() function with write ("w") or append ("a") mode to write data to a file.

```
with open("output.txt", "w") as file:
    file.write("This is some output text.")
```

**3.Logging:**

- Use the logging module for more advanced output, such as logging messages to a file or console.

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

**4.Returning Output from Functions:**

- Functions can return values that can be used elsewhere in the program.

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

---

## Control FLow

```
#=====================================Control Flow ================================
# Ternary Operator for Concise Logic:
is_raining = True
activity = "Stay indoors" if is_raining else "Go outside"
print(activity)
            #=============================================================#
# matchcase
status = 404
match status:
    case 200:
        print("Success")
    case 404:
        print("Not Found")
    case _:
        print("Unknown status")
            #=============================================
def process_value(value):
    if value == 1:
        print("One")
    elif value == 2:
        print("Two")
    elif isinstance(value, list) and len(value) == 2:
        print(f"List with two elements: {value[0]}, {value[1]}")
    else:
        print("Unknown")

def process_value(value):
    match value:
        case 1:
            print("One")
        case 2:
            print("Two")
        case [x, y]:
            print(f"List with two elements: {x}, {y}")
        case _:
            print("Unknown")


process_value(1)  # Output: One
process_value([3, 4])  # Output: List with two elements: 3, 4
# Syntax for Multiple Patterns with |
# match value:
#     case pattern1 | pattern2 | pattern3:
         # Code to execute if value matches any of the patterns

fruit = "apple"

match fruit:
    case "apple" | "banana":
        print("It's a common fruit")
    case "kiwi" | "mango":
        print("It's an exotic fruit")
    case _:
        print("Unknown fruit")

data = [1, 2]

match data:
    case [1, 2] | [3, 4]:
        print("The list matches [1, 2] or [3, 4]")
    case _:
        print("The list does not match any pattern")

# Syntax for Guards in match-case :
    # case pattern if condition:
        # Code to execute if pattern matches and condition is true
#Example: Using Guards to Check num > 0
num = 10
match num:
    case n if n > 0:
        print(f"{n} is positive")
    case n if n < 0:
        print(f"{n} is negative")
    case _:
        print("The number is zero")
#Matching Lists with Guards
data = [1, 2, 3]
match data:
    case [x, y, z] if x > 0 and y > 0 and z > 0:
        print("All elements are positive")
    case [x, y, z] if x < 0 or y < 0 or z < 0:
        print("At least one element is negative")
    case _:
        print("Unknown pattern")

#========================================Loop=========================#
print("\n===============for i in [0,1,2,3,4]=================")
for i in [0,1,2,3,4] :
    print(i,end=" ")
print("\n===============for i in range(5)=================")
for i in range(5) :
    print(i,end=" ")
print("\n===============for i in range(5,10) :=================")
for i in range(5,10) :
    print(i,end=" ")
print("\n===============for i in range(5,10) :=================")
for i in range(5,10,2) :
    print(i,end=" ")

print()

#Loop Index and Value
# ===============================
l=['mina','Ahmed','Ali']
#Basic Syntax
# ===============================
for i in range(0,len(l)):
    print(i,l[i])

#Advanced Syntax
# ===============================
for i in enumerate(l):
    print(i)
    #(0, 'mina')
    #(1, 'Ahmed')
    #(2, 'Ali')
for index,value in enumerate(l): #unpacking
    print(index,value)

# =================================================================
# for - else
the_prime_numbers=[]
for i in range(2,102):
    for j in range(2,i):
        if i %j ==0:
            break
    else :
        the_prime_numbers.append(i)

print(the_prime_numbers)
```