Python Fundamentals

Language fundamentals and syntax

- keywords
- Identifiers—names of variables,functions etc
- comments and multi line comments
- Indentation
- statements
- variables
- storage location(id())
- type of data types
- operators in python
- conditional statements
- Looping in python
- switch statements

Python Keywords

- **Keywords** = reserved words in Python.
- Cannot be used as **variable names**, **function names**, or any **identifier**.
- **Case-sensitive**.

```
import keyword
print(keyword.kwlist)
print("\nTotal number of keywords: ", len(keyword.kwlist))
```

🔶 Total number of keywords (Python 3.6): **33**

Examples: `'False', 'None', 'True', 'if', 'else', 'for', 'import', ...`

🏷️ Identifiers

- Names for entities like **classes**, **functions**, **variables**.
- **Rules**:
  1. Can include letters (a-z, A-Z), digits (0-9), and underscore `_`.
  2. **Cannot start with a digit**.
     - ❌ `12abc` → Invalid
     - ✅ `abc12` → Valid
  3. **Keywords not allowed** as identifiers.
     - ❌ `global = 1` → Invalid

```
12_abc = 12     # ❌ invalid
a_12_bc = 12    # ✅ valid
global = 1      # ❌ invalid
a@ = 10         # ❌ invalid (special symbols not allowed)
```

❌ Invalid special characters in identifiers: `@`, `!`, `#`, `$`, `%`, etc.

Python Comments

- Ignored by compilers/interpreters.
- Make code readable & explain logic.

```
# Print Hello, world to console
print("Hello, world")
```

Multi-line Comments:

1. Using multiple `#`:

```
# This is a long comment
# and it extends
# Multiple lines
```

2. Using **triple quotes**:

```
"""This is also a
perfect example of
multi-line comments"""
```

⬅️ Python Indentation

- Used instead of `{}` to define code blocks (unlike C/C++).
- Must be **consistent**.
- **4 spaces** are preferred.

```
for i in range(10):  # valid
    print(i)
```

```
for i in range(10):  # error due to incorrect indent
    print(i)
    print(i*2)
print(100)  # ❌ IndentationError
```

✅ Indentation can be skipped in **line continuation**, but avoid it.

Python Statements

- A statement ends with a newline. To continue a statement across lines, use `\` or parentheses.

```
# Using line continuation
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8
print(a)  # Output: 36
```

```
a = (1 + 2 + 3 +
     4 + 5 + 6 +
     7 + 8)
print(a)  # Output: 36
```

- Multiple statements in one line using `;`

```
a = 10; b = 20; c = 30  # put multiple statements in a single line
```

Variables

- Variables store data in memory.
- No need to declare before use.
- No need to specify data type (it is inferred automatically).

Assignments

```
a = 10
b = 5.5
c = "ML"
```

Multiple Assignments

```
a, b, c = 10, 5.5, "ML"
a = b = c = "AI"  # same value to multiple variables
```

Storage Locations

- Use `id()` to get the memory address of a variable.

```
x = 3
print(id(x))

y = 3
print(id(y))
# x and y point to the same memory location
```

```
y = 2
print(id(y))  # different memory location now
```

Data Types

- Every value has a data type (everything is an object).
- Use `type()` to get the type, `isinstance()` to check if a variable belongs to a type.

Examples

```
a = 5
print(a, "is of type", type(a))  # int
```

```
a = 2.5
print(a, "is of type", type(a))  # float
```

```
a = 1 + 2j
print(a, "is complex number?")
print(isinstance(1 + 2j, complex))  # True
print(type(a))  # <class 'complex'>
```

Boolean

- `True` and `False` are boolean values.

```
a = True
print(type(a))  # bool
```

Operators in python

1. Arithmetic Operators
2. Comparision Operators
3. Logical Operators
4. Bitwise operators
5. Assignment operators

examples :

```
a = 10
b = 4
```

```
print("Addition (a + b):", a + b)
print("Subtraction (a - b):", a - b)
print("Multiplication (a * b):", a * b)
print("Division (a / b):", a / b)
print("Floor Division (a // b):", a // b)
print("Modulus (a % b):", a % b)
print("Exponentiation (a ** b):", a ** b)
```

```
print("Equal (a == b):", a == b)
print("Not Equal (a != b):", a != b)
print("Greater Than (a > b):", a > b)
print("Less Than (a < b):", a < b)
print("Greater Than or Equal (a >= b):", a >= b)
print("Less Than or Equal (a <= b):", a <= b)
```

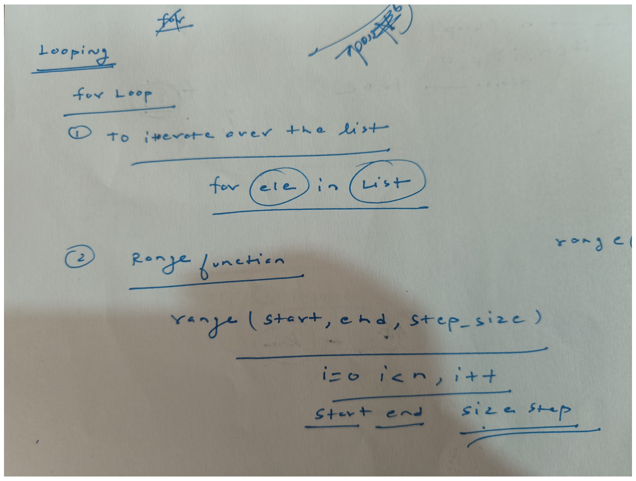
```
x = True
y = False

print("AND (x and y):", x and y)
print("OR (x or y):", x or y)
print("NOT (not x):", not x)
```

```
a = 5  # 0101 in binary
b = 3  # 0011 in binary

print("AND (a & b):", a & b)
print("OR (a | b):", a | b)
print("XOR (a ^ b):", a ^ b)
print("NOT (~a):", ~a)
print("Left Shift (a << 1):", a << 1)
print("Right Shift (a >> 1):", a >> 1)
```

```
c = 7
print("Initial value of c:", c)
c += 3
print("After c += 3:", c)
c -= 2
print("After c -= 2:", c)
c *= 2
print("After c *= 2:", c)
c /= 3
print("After c /= 3:", c)
c %= 2
print("After c %= 2:", c)
```

```
Conditional statements in python

n= int(input())
if n > 0:
    print("positive")
elif n == 0:
    print("Zero")
else:
    print("Negative")
```

Looping in python


~~for~~

Looping

for Loop

① To iterate over the list

for ele in List

② Range function

range(start, end, step_size)

i=0 i<n, i++

start end size step

range(

```
Direct Iteration

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

```
Range Function
range(start, stop, step)

for i in range(5):
    print(i)

for i in range(1, 11, 2):
    print(i)
```

Some Examples

```
# Sum of Even Numbers
number = int(input())
even_sum = 0
```

```
count = 1
while count <= number:
    if count % 2 == 0:
        even_sum = even_sum + count
    count = count +1

print(even_sum)
```

```
# Primality check
n = int(input())
count = 2
flag = False
while count <= n-1:
    if n % count == 0:
        flag = True
        break
    count += 1

if flag:
    print("n is not prime number")
else:
    print("n is prime number")
```

```
n = int(input("Enter a number: "))

d = 2
while d <= n:
    isprime = True

    count = 2
    while count * count <= d:
        if d % count == 0:
            isprime = False
            break
        count += 1

    if isprime:  # Print the number if it is prime
        print(d)

    d += 1
```

Data structures in python

Strings

- Indexing
- Slicing
- Deletion
- Immutable
- Concatenation and Multiplication
- Looping
- membership test
- Imp string methods

✅ What is a String?

A **string** is a sequence of Unicode characters enclosed in `'single'`, `"double"`, or `'''triple quotes'''`.

```
s = "Hello, World!"
```

1️⃣ Indexing and Slicing

🔷 Indexing (zero-based)

```
s = "Python"
print(s[0])   # P
print(s[-1])  # n (last character)
```

🔷 Slicing

```
print(s[1:4])   # yth
print(s[:3])    # Pyt
print(s[::2])   # Pto (step slicing)

[start:end:step_size]
```

2️⃣ Strings Are Immutable

You **cannot change** characters in a string:

```
s = "hello"
# s[0] = 'H'   Error: Strings are immutable
```

3️⃣ Deletion

You can't delete part of a string, but you can delete the **entire** string:

```
s = "hello"
del s   # ✅ Deletes the whole string variable
```

4️⃣ Concatenation and Multiplication

```
# Concatenation
a = "Hello"
b = "World"
print(a + " " + b)  # Hello World

# Multiplication
print("Hi! " * 3)   # Hi! Hi! Hi!
```

5️⃣ Looping Through a String

Same as in lists:

```
for char in "cat":
    print(char)
```

6️⃣ Membership Test

```


"th" in "python"     # True
"x" not in "python"  # True
```

7️⃣ Useful String Methods

Method	Description	Example
`upper()`	Converts to uppercase	`"hi".upper()` → `'HI'`
`lower()`	Converts to lowercase	`"Hi".lower()` → `'hi'`
`split()`	Splits string into list	`"a,b,c".split(',')` → `['a','b','c']`
`join()`	Joins list into string	`' '.join(['Hello','World'])` → `'Hello World'`
`find()`	Finds first index of substring	`"hello".find('l')` → `2`
`replace()`	Replaces part of the string	`"hello".replace('l','x')` → `'hexxo'`

Lists

✅ What is a List?

A **list** is a **sequential**, **mutable** data structure in Python.

Defined using **square brackets** `[]`.

```
fruits = ["apple", "banana", "cherry"]
```

🔁 Mutable

Lists can be **modified** — you can change, add, or remove elements.

```
fruits[0] = "grape"
print(fruits)  # ['grape', 'banana', 'cherry']
```

📏 Length of a List

```
numbers = [1, 2, 3]
print(len(numbers))  # 3
```

➕ Adding Elements

🔷 `append()` – Adds to the end

```
numbers.append(4)
# [1, 2, 3, 4]
```

🔷 `insert(index, value)` – Inserts at a specific position

```
numbers.insert(1, 10)
# [1, 10, 2, 3, 4]
```

➖ Removing Elements

🔷 `remove(value)` – Removes first occurrence

```
numbers = [1, 2, 3, 2]
numbers.remove(2)
# [1, 3, 2]
```

🔷 `pop()` – Deletes and returns the element

```
last = numbers.pop()
# last = 2, numbers = [1, 3]
```

🔷 `del` – Deletes element by index (no return)

```
del numbers[1]
# [1]
```

🔗 Extending a List

```
a = [1, 2]
b = [3, 4]
a.extend(b)
# a = [1, 2, 3, 4]
```

✅ Membership Test

```
3 in [1, 2, 3]      # True
5 not in [1, 2, 3]  # True
```

🔄 Reverse a List

```
nums = [1, 2, 3]
nums.reverse()
# [3, 2, 1]
```

🔢 Sorting a List

🔷 `sorted(list)` – Returns a new sorted list

```
sorted([3, 1, 2])  # [1, 2, 3]
```

🔷 `list.sort()` – Sorts in place

```
nums = [3, 2, 1]
```

```
nums.sort()
# [1, 2, 3]
```

🔁 Counting Elements

```
[1, 2, 2, 3].count(2)  # 2
```

🔎 Indexing & Slicing

```
lst = [10, 20, 30, 40]

# Indexing
print(lst[0])   # 10
print(lst[-1])  # 40

# Slicing
print(lst[1:3])  # [20, 30]
print(lst[::-1]) # [40, 30, 20, 10]
```

🧵 Split String to List

```
text = "a,b,c"
lst = text.split(",")
# ['a', 'b', 'c']
```

🔁 List Comprehension

🔷 Squares of 1 to 10

```
squares = [x**2 for x in range(1, 11)]
# [1, 4, 9, 16, ..., 100]
```

🔷 Cubes of 1 to 10

```
cubes = [x**3 for x in range(1, 11)]
# [1, 8, 27, ..., 1000]
```

Tuples

✅ What is a Tuple?

- A **tuple** is a **sequence of elements** like a list.
- It is defined using **parentheses** `()`.
- Tuples are **immutable** — they **cannot be changed** after creation.

```
t = (10, 20, 30)
```

🔐 Immutable

```
t = (1, 2, 3)
# t[0] = 100  ❌ Error: 'tuple' object does not support item assignment
```

🔁 Tuple Supports Most List Operations

1️⃣ Indexing and Slicing

```
t = (10, 20, 30, 40)

print(t[0])      # 10
print(t[-1])     # 40
print(t[1:3])    # (20, 30)
```

2️⃣ Length of Tuple

```
len(t)   # 4
```

3️⃣ Membership Test

```
20 in t        # True
50 not in t    # True
```

4️⃣ Count and Index

```
t = (1, 2, 2, 3)

t.count(2)     # 2 (how many times 2 occurs)
t.index(3)     # 3 (position of 3)
```

5️⃣ Sorting a Tuple

Tuples are immutable, so sorting returns a **new list**, not a tuple.

```
t = (3, 1, 2)
```

```
sorted(t)      # [1, 2, 3]
```

If you want it back as a tuple:

```
tuple(sorted(t))  # (1, 2, 3)
```

6️⃣ Mathematical Operations

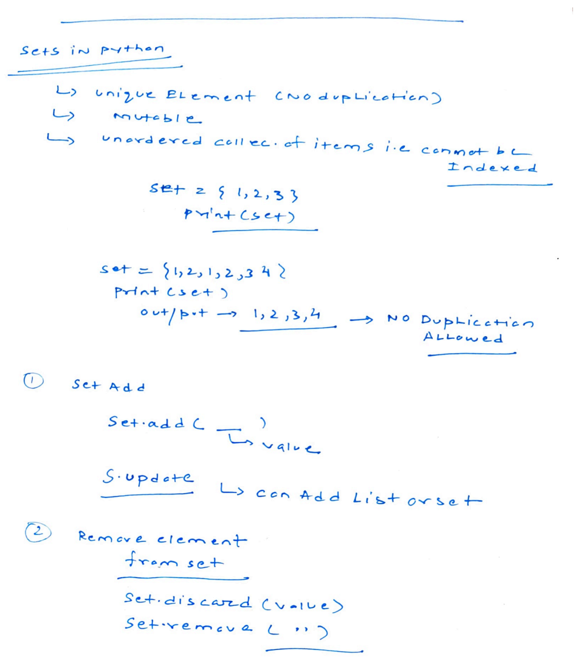
```
t = (5, 10, 15)

min(t)    # 5
max(t)    # 15
sum(t)    # 30
```

7️⃣ Tuple Deletion

- You **cannot delete individual elements**, but you can delete the **whole tuple**.

```
t = (1, 2, 3)
del t     # ✅ Deletes the entire tuple
# del t[0] ❌ Error: 'tuple' object doesn't support deletion
```

Sets



Sets in python

↳ unique Element (No duplication)
↳ mutable
↳ unordered collec. of items i.e cannot be Indexed

```
set = { 1,2,3 }
print(set)
```

```
set = {1,2,1,2,3 4 }
print (set)
out/put → 1,2,3,4
```

→ No Duplication Allowed

① Set Add

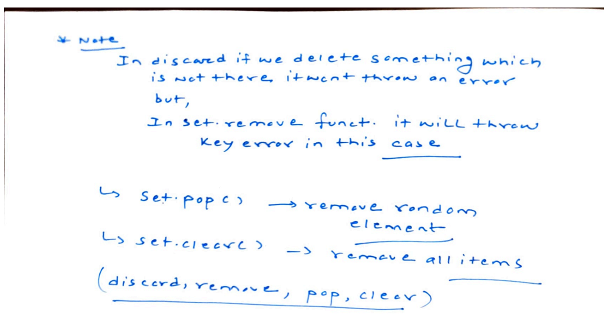
```
Set.add( __ )
```

↳ value

S.update ↳ can Add List or set

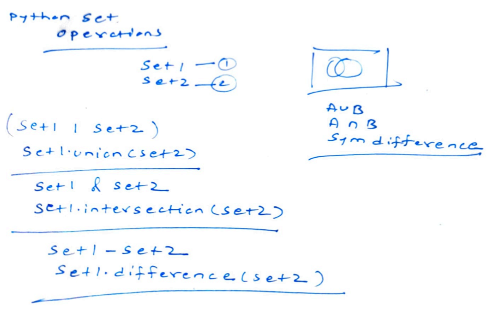
② Remove element from set

```
Set.discard (value)
Set.remove ( .. )
```



* Note
In discard if we delete something which is not there, it wont throw an error
but,
In set.remove funct. it will throw key error in this case

↳ Set.pop() → remove random element
↳ set.clear() → remove all items
(discard, remove, pop, clear)



Python set operations

Set1 — ①
Set2 — ②

A∪B
A∩B
Sym difference

```
(Set1 | Set2)
Set1.union(set2)

Set1 & set2
Set1.intersection(set2)

Set1 - Set2
Set1.difference(set2)
```

Dictionary

- Key-Value Pair Storage

```
# Dictionary with key-value pairs
student = {"name": "Alice", "age": 20, "grade": "A"}
```

- Unordered Nature

```
# Dictionaries are unordered (order may vary)
d1 = {"a": 1, "b": 2}
d2 = {"b": 2, "a": 1}
print(d1 == d2)  # True, despite different order
```

- Adding, Updating, and Accessing Elements using key

```
person = {"name": "John"}
# Adding new key-value pair
person["age"] = 30
# Updating existing value
person["name"] = "John Smith"
# Accessing value
print(person["name"])  # John Smith
```

- Deleting Elements

```
car = {"make": "Toyota", "model": "Corolla", "year": 2020}
# Delete specific key
del car["year"]
# Clear all items
car.clear()
# Delete entire dictionary
del car
```

- Dictionary Methods

```
menu = {"coffee": 2.5, "tea": 1.5}
# Common methods
print(menu.keys())    # dict_keys(['coffee', 'tea'])
print(menu.values())  # dict_values([2.5, 1.5])
print(menu.items())   # dict_items([('coffee', 2.5), ('tea', 1.5)])
print(menu.get("juice", "Not available"))  # Not available
```

- Looping Through a Dictionary

```
prices = {"apple": 0.5, "banana": 0.3, "orange": 0.4}
# Loop through keys
for fruit in prices:
    print(fruit)
# Loop through key-value pairs
for fruit, price in prices.items():
    print(f"{fruit}: ${price}")
```

- Dictionary Comprehension

```
# Create dictionary of squares
squares = {x: x**2 for x in range(5)}
print(squares)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Conditional dictionary comprehension
even_squares = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares)  # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Method	Description	Example
`.copy()`	Returns a **shallow copy**	`d2 = info.copy()`
`.get(key)`	Returns value or `None` if key not found	`info.get("age")`
`.keys()`	Returns all keys	`list(info.keys())`
`.values()`	Returns all values	`list(info.values())`
`.items()`	Returns key-value pairs as tuples	`list(info.items())`