

Looping Control Statements:-

These are the statements which is used to perform some task repeatedly.

a. WHILE LOOP:-

- Used to perform some task again and again until the given condition satisfied.
- No. of iterations are not known
- It will not works with set and dictionary
- **Syntax:**

Initialization

while condition:

Updation

- Initialization & Updation are mandatory.
- Example 1:- While loop **without updation**

```
i = 1    # Initialization
```

```
while i <= 5:
```

```
    print(i)
```

- Example 2:- While loop **with updation**

```
i = 1    # Initialization
```

```
while i <= 5:
```

```
    print(i)
```

```
i = i + 1 # Updation
```

b. FOR LOOP:-

- For loops works with any of the collection datatype
- By default, for loops go till the length of the collection and iterate over each values.
- **Syntax:-**

```
for var in collection:
```

- **Examples (for loop)**

```
for i in range(1, 6):  
    print(i)
```

```
for ch in "Python":  
    print(ch)
```

- **range():-**

1. range() is used to generate a sequence of numbers.
2. Does **not store values**, it generates them one by one

Syntax: range(start, stop, step)

- start → starting value (default = 0)
- stop → ending value (excluded)
- step → increment/decrement (default = 1)

Examples (range):-

```
print(list(range(5)))
print(list(range(1, 10, 2)))
```

Transfer Control Statements:-

1. break :	Terminated the execution of a loop
2. continue	Used to skip rest of the code
3. pass	Used as a null statement when no action is required

Code for break:-

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

Code for continue-

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Code for pass:-

```
def main():
    pass
```

Output Statements:-

print() function is used to display the output on the console.

Syntax print(val1/var, val2, val3, ..., valn, sep=' ', end='\n')

Parameters:

- **sep** → Separates multiple values (default is space ' ')
- **end** → Printed at the end of output (default is new line '\n')

Example 1: Using sep

```
print(2, 3, 4, sep='@')
```

```
2@3@4
```

```
Print(100, sep = "#")
```

```
100
```

Example 2: Using end

```
print(1, 2, 3, 4, 5, end='#')
```

```
1, 2, 3, 4, 5#
```

Example 3: Using both sep and end

```
print(1, 2, 3, 4, sep='\t', end='#')
```

```
print(1, 2, 3, 4, sep=' ', end='#')
```

Example 4:

Example 4:

```
x = print("Hello")
print(x)
```

Important Note print() does not return any value

Comprehension:-

Comprehension is a phenomenon of creating a new collection by using fewer instructions, which increases the efficiency of the program.

Comprehension is supported for:

- List
- Set
- Dictionary

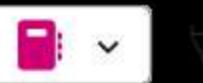
Note Tuple comprehension does not exist; it becomes a generator

Types of Comprehension

1. List Comprehension
2. Set Comprehension
3. Dictionary Comprehension

1. List Comprehension:-

It is a phenomenon of creating a new list (collection) using a single line of code.



Syntax	a. var = [exp/val for var in collection] b. var = [val for var in collection if condition] c. var = [TSB if condition else FSB for var in collection]
---------------	---

Examples:-

Print numbers from 1 to 10

```
lst = [i for i in range(1, 11)]
```

Create a list of odd numbers between 1000 and 1500

```
odd_list = [i for i in range(1000, 1501) if i % 2 != 0]
```

Square of even numbers and cube of odd numbers

```
res = [i**2 if i % 2 == 0 else i**3 for i in range(1, 11)]
```

Word-length output

```
inp = "Python is very very easy language"
```

```
# Oup = [('python', 6), ('is', 2), ('very', 4), ('easy', 4), ('language', 8)]
```

```
out = [(word.lower(), len(word)) for word in inp.split()]
```

Note: List and Set comprehensions are almost similar, except:

- List → []
- Set → {}

2. Dictionary Comprehension:-

Syntax a. var = {key: value for var in collection}

b. var = {key: value for var in collection if condition}

c. var = {key: TSB if condition else FSB for var in collection}

Examples:-

Natural numbers as keys and their squares as values

```
d = {i: i**2 for i in range(1, 101)}
```

```
d = {i: i**2 if i%2==0 else i**3 for i in range(1, 101)}
```

```
print(d)
```

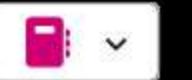
Get ASCII values of uppercase characters

```
inp = "Hai HeLLO"
```

```
# Output = {'H':72, 'L':76, 'O':79}
```

```
out = {i: ord(i) for i in inp if i.isupper()}
```

Regular Expressions (Regex):-



Regex is a technique used to search, match, and manipulate patterns in strings.

To use regex in Python, we import the **re module**.

```
import re
```

Common Regex Functions:-

search():-

1. It used to search a pattern **anywhere in the string**.
2. It returns the **first match object** if found, otherwise returns None.
3. **Syntax** `re.search(pattern, string)`

Example:

```
import re
s = "Python is very easy"
re.search("easy", s)      # span=(15, 19), match='easy'
```

match():-

1. It checks for the pattern only at the beginning of the string.
2. **Syntax** `re.match(pattern, string)`

Examples:

```
re.match("Python", "Python is easy")
re.match("easy", "Python is easy")
```



group() and groups():-

Ques: Why do we use m.group() or m.groups()?

1. re.search() and re.match() do not return the matched value directly.
2. They return a **match object** (m).
3. To extract the actual matched data, we use:
 - a. **group()** → single match
 - b. **groups()** → multiple sub-matches

group():-

1. It is used to **return the matched pattern**.
2. **Syntax** match_object.group()

Example:

```
import re
m = re.search(r"\d+", "Age is 21") # m stores match details
m.group() # extracts the first actual matched value
```

groups():-

1. It is used to return **all captured sub-groups** in the form of a **tuple**.
2. **Syntax** match_object.groups()

Example:

```
import re  
m = re.search(r"(\d+)-(\d+)", "2025-02") # () create sub-groups  
m.groups() # extracts all sub-matches together
```

Quantifiers:-

1. Quantifiers specify **how many times a character or pattern should appear.**

Quantifier	Meaning
*	0 or more times
+	1 or more times
?	0 or 1 time
{n}	Exactly n times
{n,m}	Between n and m times

2.

Examples:-

```
import re  
print(re.search(r"ab*", "a")) # * (0 or more)  
print(re.search(r"ab+", "abbb")) # + (1 or more)  
print(re.search(r"colou?r", "color")) # ? (0 or 1)  
print(re.search(r"\d{4}", "Year 2025")) # {n}  
print(re.search(r"\d{2,4}", "ID: 123")) # {n,m}  
# matches 2 to 4 digits (12, 123, 1234)  
# n = minimum times, m = maximum times
```

Functions:-

- 1. It is a name given to a memory block where the instructions are stored and which are capable of performing some specific task.
- 2. It can be utilized n no. of times after creating it.
- 3. Function are reusable block of code.

TYPES OF FUNCTIONS:-

- 1. Inbuilt Function
- 2. User - defined Function

User- defined Function:-

Syntax	<pre>def fname(args): # Statement Block return values # Function Calling fname(vals)</pre>
--------	---

Types of User- defined Functions:-

- 1. Function without args & without return values
- 2. Function with args and without return values
- 3. Function without args, with return values
- 4. Function with args and with return values

Function without args & without return values:-

Write a program to create a function that prints “Hello World”.

```
def greet():
    print("Hello World")
greet()
```

Write a program to create a function that prints the sum of two numbers(user input)

```
def add():
    a = int(input())
    b = int(input())
    print("Sum =", a + b)
add()
```

Function with args & without return values:-

1. This is a function in which passing the args is compulsory but return value is not.

Syntax	def fname(var1, var2,, varn): # Statement Block
--------	--

- 2.

	# Function Calling
--	--------------------

```
# Function Calling  
fname(val1, val2, val3, ..., valn)
```

3. No. of variables passed == No. of values passed (otherwise error)

Examples:-

Write a function that takes a name and prints a welcome message.

```
def welcome(name):  
    print("Welcome", name)
```

```
welcome("Aditya")
```

Function without args & with return values:-

Syntax **def fname():**

```
    # Statement Block  
    return val1, val2, val3, ..., valn
```

```
# Function Calling
```

```
val1, val2, val3, ..., valn = fname()
```

OR

```
var = fname()
```

Example:-

```
def get():
    a = int(input())
    b = int(input())
    return a, b
var = get()
m, n = get()
```

Function with args & with return values:-

Syntax **def fname(var1, var2, ..., varn):**
 # Statement Block
 return val1, val2, val3, ..., valn

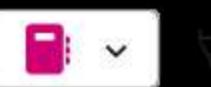
Function Calling
var = fname(val1, val2, val3, ..., valn)

Example:-

Write a program to create a function that takes two numbers and returns their sum.

```
def add(a, b):
    return a + b
result = add(10, 20)    # 10, 20 are the actual args
```

```
print(result)
```



ARGUMENTS:-

1. Formal Argument
2. Actual Argument

1. Formal Argument:-

1. Formal arguments are the **variable names written in the function definition**.
2. They act as **placeholders** for the values passed during function calling.

Example:-

```
def add(a, b):    # a, b → formal arguments  
    return a + b
```

3. Types of Formal Arguments:-

- a. Positional Arguments
- b. Default Arguments
- c. Keyword Arguments
- d. Variable Length Arguments

Positional Arguments:-

1. Values are assigned **based on position**.
2. Example:

```
def add(a, b):  
    print(a + b)  
add(20, 10)
```

3. Key Characteristics:-

- a. Order Matters
- b. Mandatory

Default Arguments:-

1. Default arguments have **default values**.
2. Default params must appears after the non-default params.
3. Order of default args doesn't matter

Syntax `def fname(var1, var2, var3,..., k1 = dv, k2 =dv,, kn = dv):`

Statement Block
return val1, val2, val3, ..., valn

4.

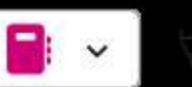
Function Calling
`var = fname(val1, val2, val3, ..., valn, k1=val, k2=val,..., kn = val)`

Example:

```
def greet(name="User"):  
    print("Hello", name)
```

`greet()`

```
greet()  
greet("Aditya")
```



...

Keyword Arguments:-

1. Keyword arguments are those in which **values are passed using the formal argument names** during function calling.
2. **Key Points:**
 1. Values are passed using **argument names (keys)**.
 2. **Order of arguments does not matter**.
 3. Improves **code readability**.
 4. Can be mixed with positional arguments, but **positional arguments must come first**.

Syntax def fname(var1, var2, var3, ..., varn):

Statement Block

return val1, val2, ..., valn

3.

Function Calling

var = fname(var1=val1, var2=val2, ..., varn=valn)

Example:-

```
def info(name, age):  
    print("Name:", name)  
    print("Age:", age)
```

```
info(age=20, name="Aditya")
```

```
info(age=20, name="Aditya")
```



Keyword arguments **must come after positional arguments.**

```
info(name="Aditya", 20) # Error
```

Variable length Arguments:-

1. Variable length arguments are used when the **number of arguments passed to a function is not fixed.**

2. Types of Variable Length Arguments

1. Non-Keyword Variable Length Arguments (*args)
2. Keyword Variable Length Arguments (**kwargs)

Non-Keyword Variable Length Arguments (*args):-

1. Multiple values are passed **without using keywords** and are received as a **tuple**.
2. Order of values matters
3. Used when argument count is unknown

Example:

```
def total(*n):  
    print(n)  
    print("Sum =", sum(n))
```

```
total(10, 20, 30, 40)
```

Keyword Variable Length Arguments (**kwargs):-

1. values are passed **using keywords** and are received as a **dictionary**.
2. **Order does not matter**
3. Used when both **keys and values** are required

Example:-

```
def details(**data):  
    print(data)  
details(name="Aman", age=41, city="Prayagraj")
```

Lambda Function:-

1. A lambda function is a **small, anonymous function**.
2. It contains **only a single expression**.
3. It is defined using the **lambda keyword**.
4. Mostly used for **short and simple operations**.

5. **Syntax**

- a. var = lambda args : expression
- b. var = lambda args : TSB if condition else FSB

Example:-

Using Function:

```
def even_odd(n):  
    if n % 2 == 0:  
        print("Even")  
    else:  
        print("Odd")  
even_odd(10)
```

Using Lambda Function:

```
even_odd = Lambda n: "Even" if n % 2 == 0 else "Odd"  
print(even_odd(5))
```

QUESTIONS USING LAMBDA

1. WAP to check whether the given data is float or not

```
is_float = Lambda x: "Float" if type(x) == float else "Not Float"  
print(is_float(10.5))
```

2. WAP to find the sum of 3 numbers

```
sum3 = Lambda a, b, c: a + b + c  
print(sum3(10, 20, 30))
```

Module & Packages:-

Module:-

1. A module is a **file** that contains **functions, variables, and classes**, which can be reused in other programs.
2. For Example:-

Step 1: Create a Python file ==> mymath.py

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

Step 2: Import the module in another file

Step 2: Import the module in another file

```
import mymath  
print(mymath.add(10, 5))  
print(mymath.sub(10, 5))
```

3. Ways to Import a Module:-

- a. import module_name
- b. from module_name import member
- c. import module_name as alias

Package:-

1. A package is a **collection of related modules** stored inside a **folder (directory)**.

2. Creating a Package:-

Folder Structure:

```
mypackage/  
└── __init__.py  
└── add.py  
└── mul.py
```

add.py

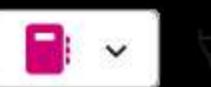
```
def add(a, b):  
    return a + b
```

mul.py

```
def mul(a, b):
```

```
def mul(a, b):  
    return a * b
```

...



Using Package Modules

```
from mypackage import add, mul  
print(add.add(10, 20))  
print(mul.mul(5, 4))
```

__init__.py File

Purpose:

- Marks a directory as a **package**
- Executes when package is imported

Scope of Variables:-

1. Global Variables

Global variables are variables that are created **outside any function** (in the main program).

Key Points:

- Created **outside** the function
- Can be **accessed (read)** inside a function
- **Cannot be modified** inside a function directly
- To modify them inside a function, we must use the **global keyword**

Example 1: Accessing a Global Variable

```
x = 10 # global variable
```