MODULE-3

Selection and Itration using python

1. The if statement (single-way selection)

This statements allows you to execute a block of statement only if a particular condition is true.

Syntax:

```
If <condin>:
  # body of if statement
```

example

(i.)
```
num = 10
If num > 0
    Print ("number is greater than zero")
Print (" statement outside if.")
```

(ii.)
```
x = int(input ("enter a number"))
if x > 5
    Print (f"{x} is greater than 5")
```

2. The If else statement (two-way selection)

The Ifelse statement adds an alternative path, executing one block of code if the condition is true, and another if the condith is false.

Syntax:

```
If <condition>:
  # block of code if condith is true.
```

```
else:
    # block of code if condition is false.
```

example:

```
x = int(input("enter a number"))
if x > 5
    Print (f"{x} is greater than 5")
else
    Print (f"{x} is less than 5")
```

3. The elif statement (multy-way select^n)

The elif (elseif) statement allows you to check multiple conditions sequentially. As soon as one condition is true, the corresponding block of code executes, and the remaining conditions are skipped.

Syntax

```
If <condition> 1:
    # code if condition 1 is true.
elif condition 2:
    # code if condition 2 is true.
else:
    # code if neither condition 1 nor condition 2 is true.
```

example:

```
x = int (input ("enter a number"))
If  x > 0
    Print (f"{x} is a positive number")
elif  x < 0
    Print (f"{x} is a negative number")
else  x = 0
    Print (f"{x} is equal to zero").
```

```
x=int(Input("enter first number"))
y=int(Input("enter second number"))
z=int(Input("enter third number"))

If  x > y,
    Print (f"{x} is larger than y")
elif  x < y and z < y
    Print (f"{y} is larger than x")
else
    Print (f"{z} is largest")
```

Iteration in Python

1. definite Iteration
2. Indefinite Iteration

For loop

Syntax: ⓘ ← loop header

```
for <variable> in range(<an integer expression>):
    <statement 1>
    ⋮
    <statement n>
```

} loop body.

Note:
Python range() function

(i.) The range() function returns a sequence of numbers, mainly used when working with for loops.

(ii.) The range() function can be represented in 3 different ways.

range(stop_value) - starting from 0 by default, an increments by 1 (by default) upto stop value.
range(start_value, stop_value) - This generates sequence based on the start and stop value increments by 1 (by default)
range(start_value, stop_value, step_size) - It generates the sequence by incrementing the start value using the step size until it reaches the stop value.

Q. Create a sequence of numbers from 0 to 5 and print each item in the sequence.

```
for i in range(6)
    print(i).
```

output = 0 1 2 3 4 5

Q. Create and Print sequence of numbers from 3-5.

```
for i in range(3,6)
    Print(i)
```

Create and print numbers from 3-19, increment by 2.

```
for i in range(3,20,2)
    Print(i)
```

* for i in range(4)
    Print(i, end=" ")

Output = 0 1 2 3

While loop

Syntax:

```
while <condition>:
    <sequence of statement>
```

example:

```
count = 0
while count < 5 :
    Print(count)
    count = count+1
```

0 1 2 3 4

count += 1
count = count+1

Q. Find the sum of first 100 numbers using for loop as well as while loop.

for loop:

```
sum = 0
for count in range(1,101) :
    sum = sum + count
Print(sum)
```

first step: (0+1)

while loop:

```
sum = 0
count = 1
while count <= 100
    sum = sum + count
    count += 1
Print(sum)
```

c = 1,
s = 0+1 = 1
c = 2
s = 1+2 = 3

Loop control statements

Python provides 3 loop control statements that control the flow of execution in a loop.

(i) "Break" statement
(ii) "Continue" statement
(iii) "Pass" statement

(i) Break

```
for number in range (1,10):
    If number == 5 :
        break
    Print (number)
```

Output : 1
2
3
4

→ while loop:

```
number = 1
while number <= 10
    If number == 5 :
        break
    Print (number)
    number = number+1
```

(ii) Continue

→ for loop

```
for number in range (1,10):
    If number == 6
        continue
    Print (number)
```

Output : 1
2
3
4
5
7
8
9

The Break statement is used to immediately exit a loop even if the loop condition is still true.

The continuos statement is used to skip the rest of the code inside the current loop iteration and move on to the next iteration of the loop.

(iii) Pass

The pass statement in python is a placeholde used when a statement is required syntactically but no action is needed.

Q. function to check if a number is positive or negative and do nothing if it is zero.

```
num = int (input ("enter a number"))
If num > 0 :
    Print (num, " is a positive number")
elif num < 0 :
    Print (f" {num} is a negative number")
else
    Pass
Print ("outside")
```

Eg: enter a number 0
outside.

Nested Loop in Python

Loops inside other loops; ~~it mea~~

```
for variable 1 in outer sequence.
    # code block for the outer loop.
    for variable 2 in inner sequence.
        # code block for the inner loop.
```

Q. Write a Program to print the following Pattern.

```
*
* *
* * *
* * * *
```

Print x instead for i=1 => 1,0 (n,n-1)

```
→ for i in range (1,5):
      for j in range (i):
          Print("*", end = " ")
      Print ()
```

Python datatypes:

1) Numeric – int, comple no, float
2) dictionary
3) Boolean
4) Set – sb
5) Sequence type – string, list, tuple

String indexing

Positive indexing → 0,1,2,3,4
Negative indexing → -5,-4,-3,-2,-1

Eg: HELLO

```
Str = Hello
Print (str [1])
```

Output = e.

String traversal

```
str = "Hello"
for i in str:
    Print(i)
```

Output :
H
E
L
L
O

String concatenation and Repetation

```
str = "py"
str 2 = "thon"
Print (str + str 2)
```

Output = python

```
str = "python"
Print(string * 3)
```

Output = Pythonpythonpython

String slicing

```
text = "Hello world"
Print (text [0:5])
```

Out = Hello

Slicing [n:m]
Start printing from n upto m-1.

→ Print (text [:5])
Out = Hello
→ Print (text [7:])
Out = World
} omitting indexes.

→ Print (text [-6:])
Out = World

→ Print (text [0:5:2])
Out = HLO

0 : 5 : 2
Starting index — ending index — step size

str. lower ()
→ str = " HELLO "
Print (str. lower ())
Out = hello

str. Upper ()
→ str = "hello"
Print (str. upper ())
Out = HELLO

str. replace (old, new)
→ str = "hello world"
str. replace ("Hello", "Hi")
Out = Hi world.

str = " Hello "
Print (str. find ("e"))
Output = 1.

Print (str. find ("g"))
Output = -1

Print (str. find ("world"))
Output = 6

str = "hello"
Print (str. count ("l"))
Output = 2

str = "hello world"
Print (str. capitalize ())
Output - Hello world

str = "1234"
Print (str. isdigit())
Out = True

str = "hello world"
Print (str. split ())
Out - 'Hello' 'world'

→ List:
["a", "b", 1, 2, 3] a = Hello
a[0] = a

newlist = [10, "hi", 9, "a"]
Print (newlist)
Output => 10, hi, 9, a

my list = [1, 2, "a", 'b']
for i in my list
Print (i)

list = [x for x in range(5)]
Print (list)
Out = 0
1
2
3
4

```
list = [x*x for x in range(5)]
Print (list)
```

0
1
4
9
16

numbers which ÷ 2 get 0. upto 10.

```
list = [x for x in range (10) if x%2=0]
Print (list)
```

0
2
4
6
8

→ List concatenation

```
Prime = [2,3,5,7]
Composite = [4,6,8,10]
num = prime + composite
Print (num)
```

2 3 5 7 4 6 8 10

→ (* Operator)

```
binary = [0,1]
byte = binary * 4
Print (byte)
```

0 1 0 1 0 1 0 1

```
a = [3,4,5]
b = [3,4,5]
Print (a==b)
```

→ True

```
a = [2,3,7]
b = [3,1,5]
Print (a<b)
```

Output = false

```
a = [2,3,7]
b = [3,3,9]
Print (a<b)
```

Output = True

```
a = [2,3,7]
b = [2,4,6]
Print (a>b)
```

Out = false

```
even = [2,4,6,8]
Print (2 in even)
```

Out = True

```
composite = [4,6,8]
Print (2 not in composite)
```

Out = True

```
list = [1,2,3,"a","b","c"]
sub = list [1:3]
Print (sub)
```

Out = 2 3

assume 0 if nothing.
[:4] → [0:4] → 1,2,3,"a"
[:] => [0:6]

→ List Mutation

~~a = [1,2,4,6]~~

```
even = [2,4,6,8]
even [3] = [8,10]
Print (even)
```

```
even = [1,1,1]
even [3:3] = [5,6]
Print (even)
```

Out = 1, 1, 1, 5, 6

```
even = [2,4,6,8]
even [3] = [8,10]
Print (even)
```

Out = 2,4,6,8,10.

even = [2, 4, 6, 8]

even [2:3] = []

Print (even)

Out = 2, 4, 8

del even [2]

Print (even)

Out = 2, 4, 8

∵ 2 : 3-1

2 : 2

list index (element)

List [1, 2, "hello", "hi"]

Print (list. index ("hello"))

Out → 2.

~~Insert~~ list. insert (position, element)

List. insert (2, "python")

Print (list)

Out → [1, 2, "python", "hello", "hi"]

list. append (element)

adds element at the end

list = [1, 2, "hello", "hi"]

list. append (8)

Print (list)

Out → [1, 2, "hello", "hi", 8]

list. remove (element)

list. remove (2)

Print (list)

Out → [1, "hello", "hi", 8]

list. Pop ()

removes and returns last element.

list = [1, 2, "hello", "hi"]

Print (list. pop ())

print (list)

→ Out – hi

→ [1, 2, "hello"]

* list. sort ()

list = [1, 12, 30, 2, 40]

list. sort ()

Print (list)

Out – [1, 2, 12, 30, 40]

list. reverse ()

list = [1, 2, "hello", "hi"]

list. reverse ()

Print (list)

Out – ["hi", "hello", 2, 1]

a = 'python'

list_1 = list (a)

Print (list_1)

Out – ['P', 'y', 't', 'h', 'o', 'n']

↳ Convertion of word to list

List of list or nested list (list inside list)

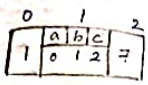
list_1 [1, ["a", 'b', 'c'], 7]

Print(list_1[0]) — out=1

Print(list_1[2]) — out=7

Print(list_1[1][0]) — out=a

Print(list_1[1][1]) — out=b



Tuple

Ordered immutable sequence of elements. (Euler's theorem)
(Can't make any change)

Heterogeneous, index ed

~~tuple~~ a = ("a", 'b', 1, 2.7, True)

Print(type(a))

Out – tuple.

If only 1 element, t_1 = ('a',)

t_1 = tuple("string")

Print(t_1)

Out – ('s', 't', 'r', 'i', 'n', 'g')

} string to tuple

t_2 = tuple([1,2,3])

print(t_2)

Out – (1,2,3)

} list to tuple

t_3 = tuple([4])

Print(t_3)

Out – (4,)

t_1=('a')

Print(type(t_1))

Out = ~~String~~

<class, 'string'>.

lan = tuple("python")

Print(lan[0])

Print(lan[1:4])

out = ('p',)

('y','t','h')

t = ('p','y','t','h','o','n')

Print(len(t))

Out = 6

In tuple, enclosis paranthesis is optional.

a,b = 5,10

Print(a,b)

a,b = b,a

Print(a,b)

out = 5,10

10,5

numpy
numerical python

numpy package and arrays

array – homogeneous collection of data items.

If you want to use the components of numpy, is import the numpy package.

Import numpy as np.

arr = np.array([1,2,3,4,5,6])

Print(arr)

out = [1,2,3,...6]

```
Mat = np.array ([1,2,3], [4,5,6])
Print (mat)
Out = [1,2,3]
      [4,5,6]

Mat = np.array ([1,2,3], [4,5,6])
Print (mat, shape)
Out = (2,3)

Mat = np.array ([1,2,3], [4,5,6])
Print (mat.ndim)
Out = 2

arr = np.array ([1,2,3,4,5,6])
Print (arr[3])
   Out = 4

Mat = np.array ([1,2,3], [4,5,6])
Print (mat[1])
   Out = [4,5,6]

Print (mat[0,2])
   Out = 3

Print (arr[-3])
   Out = 4.

Print (mat[-1,2])
```

```
for elt in arr:
    Print (elt, end = " ")
    Out: 1 2 3 ... 6
```
} array traverse

```
for row in mat:
    for elt in row
        Print (elt, end = " ")

mat = np.array ([1,2,3], [4,5,6])
mat.reshape (1,6)
    Out = [1, 2, 3, 4, 5, 6]

arr = np.array ([1,2,3,4,5,6])
arr.reshape (3,2)
    [1,2]
    [3,4]
    [5,6]
```

Python function ()

```
def function-name (parameters):
    # statement
    return expression

def greet ():
    Print ("Hello")
Out: Hello

def greeet ():
    Print ("Hello")
greet ()
    Out:
```

Advantages of function:

code reusability, modularity, ease of debugging, improved readability.

function with No arguments and return value.

```
Eg: def greet():
        Print("Hello")
```

function with arguments and parameters, no return value.

```
def welcome(name):
    Print(f"Hello {name}")
welcome(arjun)
```

function with argument and return value

```
def square(x):
    return(x*x)
⋮
a = Square(4)
Print(a)
```

Out: 16.

function with multiple arguments

```
def sum(a,b):
    return a+b
⋮
x = input("enter 1 No")
y = input("enter 2 No")
c = sum(x,y)
Print(c)
```

Q Write a fn to code celsius-farenheit that takes a temperature in celcius and convert it into farenheit using the formula:-

```
def celsius_to_Fahrenheit(celsius): C(9/5)+32
    fahrenheit = (celsius x 9/5) + 32
    return Fahrenheit
f = celsius_to_fahrenheit(25)
Print(f)
```

Out: 77°F

Problem decomposition

~~when a Solving~~ Involves breaking into small functions/modules

Why problem decomposition?

- Improves clarity
- ~~Speci~~ facilitates reusability
- enhances debugging
- encourages collaboration
- Promotes maintainability

Best Practices for decomposition

- keep function small.
- use meaningful names.
- test implementary
- document your code.
- think recursively

Modularisation
breaking of large programmes into reusable modules.

Module
self-contained unit of code that can be a
- function
- class
- package
- library

Why Modularisation?
- Improved readability
  - Easier to read and navigate python programmes
  - easier maintanence
  - allows you to concentrate on a specific part
- code reusability
- supports collaboration
- encourages testing
- failitates scalability

Recursion

```
def hello():
    Print("Hello")
    hello()

hello()
```

Out: Infinite hello.

```
def Hello(count):
    if count <= 0 - base case
        return
    Print("Hello")
    hello(count-1)   -> recursive case (minus)
hello(5)
```

Recursion is a programming technique where a function code itself to solve smaller instances of the same problem. It continues until it reaches a base case, which is a condition that terminates the recursive calls.

Key components of Recursion:-

(i) Base case

(ii) Recursive case

-(i) A condition to stop the recursion and prevent infinite calls.

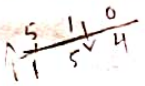
-(ii) A smaller version of same problem, leading towards the base case.

Call stack

A stack is a data structure that follows LIFO. [last In first Out] principle. The call stack is a special datastructure used by programes to manage function calls. whenever a function is called (i) the current state of fn is pushed onto the call stack.

(ii) the recursive call creates a new instance of the function, which is also pushed onto the stack.

(iii) when the base case is reached, the stack begins to unwind, resolving each fn call in reverse order.



Step	Stack	unwind. Action Taken
5	hello(1) ↓	Print("Hello"), hello(0)
4	hello(2)	Print("Hello"), hello(1)
3	hello(3)	Print("Hello"), hello(2)
2	hello(4)	Print("Hello"), hello(3)
1	↑ wind hello(5)	Print("Hello"), hello(4)

Q: Write a Program to compute the factorial of a number using recursive method.

```
def factorial(n):
    If n == 0:  or  n == 1:
        return 1
    else:
        return n * factorial(n-1).
n = int(input("enter number"))
```

Print (factorial (n))

Call stack

1. factorial (4)
   n = 4
   return 4 * factorial (3) = 24

2. factorial (3)
   n = 3
   return 3 * factorial (2) = (6)

3. factorial (2)
   n = 2
   return 2 * factorial (1) = (2)

4. factorial (1)
   n = 1
   return 1 * factorial (0) = (1)
   (1)

→ Use recursive function to add two numbers.

```
def add (a,b):
    If b == 0:
        return a
    else:
        return add (a,b-1)+1
Print (add (3,4)) OR  a = Int ("enter first number")
                      b = Int ("enter second number")
                      Print (a+b)
```

→ Use recursive function to ~~calculate the factorial of a~~ number find greatest common divisor of two +ve numbers.

```
def gcd (a,b)
    If b == 0:
        return a
    else:
        return gcd (b, a%b)
```

→ a = 48 b = 18

gcd (48,18) => gcd (18,12)
gcd (18,12) => gcd (12,6)
gcd (12,6) => gcd (6,0)
gcd (6,0) => 6

Q. Use recursion to find fibinoki series.