Functions & Exception Handling

🎯 Session Objectives:

- ✅ Understand what functions are and why we use them.
- ✅ Learn to define functions, with parameters and arguments.
- ✅ Explore the use of the return statement.
- ✅ Understand scope and namespaces.
- ✅ Understand recursion
- ✅ Use lambda (anonymous) functions
- ✅ Understand Python's exception handling model
- ✅ Apply try, except, else, finally, and raise statements effectively

```python
# Returning Sum & Products
def sum_and_product(x,y):
    return x+y , x*y

_sum , _product = sum_and_product(11,7)
print(_sum) # 18
print(_product) # 77
```

```
18
77
```

```python
def sum_and_product(x,y):
    return x+y , x*y, x-y, x/y

_sum , _product , _sub , _divide = sum_and_product(11,7)
print(_sum) # 18
print(_product) # 77
print(_sub) # 4
print(_divide) # 1.57
```

```
18
77
4
1.5714285714285714
```

```python
# Global Scope Vs Local Scope
x = 10
def greet():
    y = 15
    print(x) # 10
    print(y) # 15
greet()
```

```
10
15
```

```python
# Global Scope Vs Local Scope
x = 10
def greet():
    y = 15
    print(x) # 10
    print(y) # 15
greet()
print(x) # 10
# print(y) # NameError: name 'y' is not defined
```

```
10
15
10
```

```python
#  Local Scope
def greet():
    m = 10
    n = 15
    print(m) # 10
    print(n) # 15
greet()
# print(m) # NameError: name 'm' is not defined
# print(n) # NameError: name 'n' is not defined
```

```
10
15
```

```python
gesture = "Happy" # global
def how_you_feel():
    gesture = "worried" # Local
    print(f"I'm feeling {gesture}")

print(f"I'm feeling {gesture}") # Happy
how_you_feel() # 'worried'
print(f"I'm feeling {gesture}") # Happy
```

```
I'm feeling Happy
I'm feeling worried
I'm feeling Happy
```

```python
# What if I'll give power to local scope to make it recognize Globally [using 'global' keyword]
i = 10 # global
def greet():
    global j
    j = 15 # local
    print(i) # 10
    print(j) # 15

greet()
print(j) # 15
```

```
10
15
15
```

```python
# Nested Function [Level - 'LEGB']
p = 11
def outer_fx():
    q = 21
    def inner_fx():
        r = 51
        s = 101
        print(p) # 11 [Global]
        print(q) # 21 ['Enclosing Type']
        print(r) # 51 ['Local']
        print(s) # 101 ['Local']
        print(len(['a','b','c','d',1,False,'Coding'])) # 7 ['Built In']
    # print(p) # 11
    # print(q) # 21
    inner_fx()
outer_fx()
# print(p) # 11
```

```
11
21
51
101
7
```

```python
# Nested Function [Level - 'LEGB']
p = 11
def outer_fx():
    q = 21
    def inner_fx():
        r = 51
        s = 101
        print(p) # 11 [Global]
        print(q) # 21 ['Enclosing Type']
        print(r) # 51 ['Local']
        print(s) # 101 ['Local']
        print(len(['a','b','c','d',1,False,'Coding'])) # 7 ['Built In']
    print(p) # 11
    print(q) # 21
    # inner_fx()
outer_fx()
print(p) # 11
```

```
11
21
11
```

```python
a = 11
b = 21
def outer_fx():
    c = 51
    d = 77
    def inner_fx():
        e = 99
        global f
        f = 101
        print(e+f) # 200
    inner_fx()
    print(f) # 101
    print(b) # 21
    print(c) # 51
    # print(e) # Local Scope # NameError: name 'e' is not defined
outer_fx()
print(f) # 101
```

```
200
101
21
51
101
```

```python
# Shadowing in Nested Scope: [uvw]
u = 5
def first_layer():
    u = 15
    v = 25

    def second_layer():
        u = 35
        w = 45
        print("In Second Layer")
        print(u) # 35
        print(v) # 25
        print(w) # 45

    print("In First Layer")
    print(u) # 15
    print(v) # 25
    second_layer()
first_layer()
# print(u) # 5 [Global Scope]
```

```
In First Layer
15
25
In Second Layer
35
25
45
```

```python
u = 5
a = 7
def first_layer():
    u = a +  10 # 17
    v = 25

    def second_layer():
        u = 35
        w = 45
        print("In Second Layer")
        print(u) # 35
        print(v) # 25
        print(w) # 45

    print("In First Layer")
    print(u) # 17
    print(v) # 25
    second_layer()
first_layer()
print(u) # 5 [Global Scope]
```

```
In First Layer
17
25
In Second Layer
35
25
45
5
```

```python
u = 5
a = 7
def first_layer():
    u = a +  10 # 17
    v = u + 25 # 42

    def second_layer():
        u = 35
        w = 45
        print("In Second Layer")
        print(u) # 35
        print(v) # 42
        print(w) # 45

    print("In First Layer")
    print(u) # 17
    print(v) # 42
    second_layer()
first_layer()
print(u) # 5 [Global Scope]
```

```
In First Layer
17
42
In Second Layer
35
42
45
5
```

Python Searches in LEGB Order:

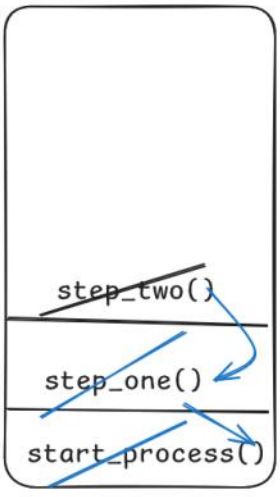
Level	Meaning
L	Local: inside current function
E	Enclosing: functions inside functions
G	Global: top-level script
B	Built-in: Python's built-in functions

```python
# Nested Function with Internal Call
def start_process():
    print("Starting the Process..")
    step_one()
    print("Main Process Complete.")

def step_one():
    print("Starting Step One")
    step_two()
    print("Step One Complete")

def step_two():
    print("Starting Step Two")
    print("Performing the Final Operations.....")
    print("Step Two Complete")

start_process()
```



Call_Stack

Console

```
Starting the Process
Starting Step One
Starting Step Two
Performing the Final Operations.....
Step Two Complete
Step One Complete
Main Process Complete
```

```
Starting the Process..
Starting Step One
Starting Step Two
Performing the Final Operations......
Step Two Complete
Step One Complete
Main Process Complete.
```

```python
# Complex Nested Functions:
def math_operations(p,q):
    def do_sum(m,n):
        return m+n
    def do_product(m,n):
        return m*n
    def execute(action,m,n):
        if action == 'sum':
            return do_sum(m,n)
        elif action == 'product':
            return do_product(m,n):
        else:
            return 'Unknown Action'
    return execute("sum",p,q) , execute("product",p,q)

result = math_operations(11,7) # (sum,product)
print(result) #(18,77)
```

Outer Function

Inner Function

18

77

Memory

result = None

p = 11

q = 7

m

n

Recursion:

What is Recursion?

Recursion is when a function calls itself to solve a smaller instance of the same problem. It consists of:

1. Base Case : Stops the recursion.
2. Recursive Case : The function calls itself with smaller inputs.

Factorial(n) → fact(n) = n * fact(n-1)

fact(n-1) = n-1 * fact(n-2)

fact(n-2) = n-2 * fact(n-3)

fact(n-3) = n-3 * fact(n-4)

fact(n-4) = n-4 * fact(n-5)

Base Case

0! = 1

1! = 1

Factorial

$$n! = n * (n-1) * (n-2) * (n-3) * \cdots * 3 * 2 * 1$$

0! = 1

1! = 1

2! = 2 x 1 = 2

3! = 3 x 2 x 1 = 6

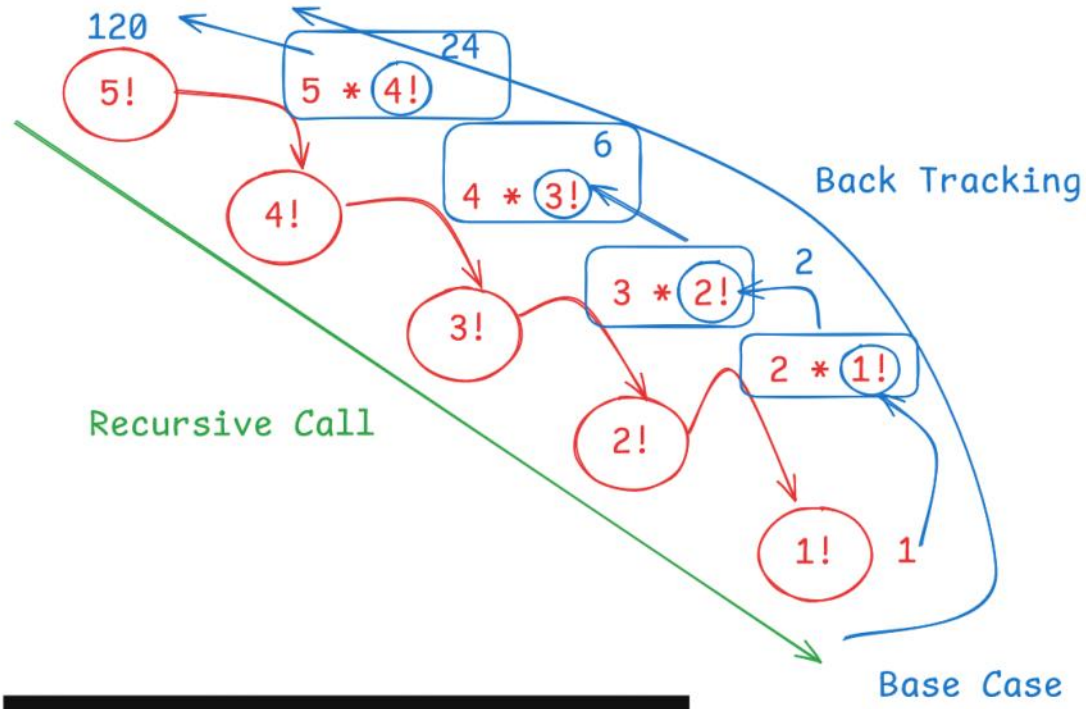
4! = 4 x 3 x 2 x 1 = 24

QG

In mathematics, PMI stands for the Principle of Mathematical Induction, a powerful deductive reasoning technique used to prove that a statement or formula holds true for all natural numbers (or any specified set of integers). It involves two main steps: first, proving the statement for a base case (usually n=1), and second, showing that if the statement is true for an arbitrary integer 'k', it must also be true for the next integer 'k+1'.

How the Principle of Mathematical Induction (PMI) Works

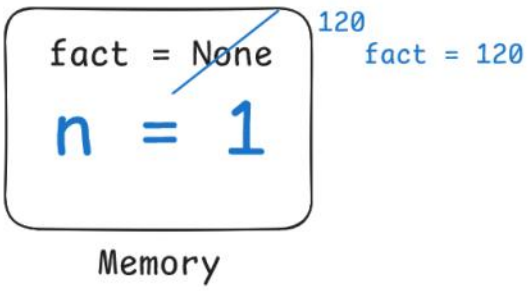
1. **Define the Statement P(n):** Clearly state the mathematical statement or formula you want to prove, involving a natural number 'n'.
2. **Base Case (P(1) is True):** Prove that the statement P(n) is true for the first natural number, usually n=1.
3. **Inductive Hypothesis (Assume P(k) is True):** Assume that the statement P(k) is true for some arbitrary, but fixed, natural number 'k'. This assumption is called the inductive hypothesis.
4. **Inductive Step (Prove P(k+1) is True):** Using the inductive hypothesis, prove that the statement P(k+1) must also be true. This demonstrates the "domino effect" – if one domino (the statement for 'k') falls, it knocks down the next one (the statement for 'k+1').
5. **Conclusion:** If both the base case and the inductive step are satisfied, then by the Principle of Mathematical Induction, the statement P(n) is true for all natural numbers n.

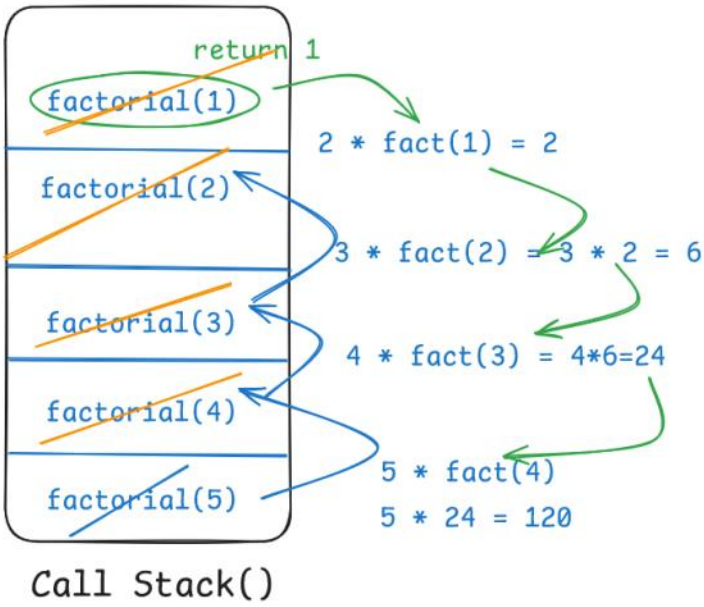



```python
def factorial(n):
    if n == 0 or n == 1: # Base Case
        return 1
    else:
        return n * factorial(n-1)

fact = factorial(5) # 120
print(fact)
```

```
120
```





Iterative Code:

```python
# Iterative Approach:
def factorial(n):
    result = 1
    for i in range(2 , n+1):
        result *=i
    return result
print(factorial(5))
```

```
120
```

What is a Lambda Function?

A lambda is a short, one-line anonymous function used for small operations without using def.

Syntax:

```
lambda arguments: expression
```

What are Exceptions?

Exceptions are errors that disrupt the flow of your program. Common ones include:

Error	Description
`SyntaxError`	Invalid code structure
`IndentationError`	Wrong indentation
`TypeError`	Wrong data type
`NameError`	Using undefined variables
`ValueError`	Invalid value
`IndexError`	Out-of-range index
`KeyError`	Missing dictionary key
`AttributeError`	Missing object method/attr
`ZeroDivisionError`	Division by 0

try-except-else-finally: Error Handling Structure

Block	Purpose
`try`	Code that might raise error
`except`	Handle specific error types
`else`	Runs if `try` succeeds
`finally`	Always runs (cleanup etc.)

try-except-else Statement

What is the else block?

The else block runs only if no exceptions are raised in the try block.

Syntax:

```
try:
    # Code that might raise an exception
except Exception1:
    # Handle Exception1
except Exception2:
    # Handle Exception2
else:
    # Runs ONLY if no exception occurs
```

try-except-finally Statement

What is the finally block?

The finally block always runs, no matter what.

Even if:

- An exception occurs
- No exception occurs
- The program is interrupted with return, break, or raise

Syntax:

```
try:
    # Risky code
except ExceptionType:
    # Handle error
finally:
    # Always run this cleanup code
```

raise Keyword

What is raise?

The raise keyword lets you intentionally trigger an exception.

Syntax:

```
raise ExceptionType("Error message")
```