🚀 Advanced Python Functions – Complete Explanation

📌 Plan of Action

In this section, we focus on advanced functional programming concepts in Python that help write **cleaner, shorter, and more efficient code**. These concepts are widely used in **data science, analytics, and backend development**.

Topics covered:

- Lambda Functions
- Map Function
- Reduce Function
- Filter Function
- Recursion

🔹 Lambda Functions

▶ What is a Lambda Function?

A **lambda function** is a small, anonymous function in Python.

- It does **not have a name**
- It is written in **one single line**
- It can take **any number of arguments**
- It returns **only one expression**

Lambda functions are mainly used when a function is needed **temporarily** and for **simple operations**.

▶ Why Use Lambda Functions?

- Reduces code length
- Improves readability for small tasks
- Commonly used with `map`, `filter`, and `reduce`
- Avoids writing full function definitions for simple logic

▶ Lambda vs Normal Function

- Normal functions are better for **complex logic**
- Lambda functions are ideal for **short, one-time operations**
- Lambda makes code look **clean and professional**

In [1]:

```
# aam zindagi
def sum(x,y):
    return(x+y)
```

In [2]:

```
# mentos zindagi
# lambda function
sum= lambda x,y : x+y
```

In [3]:

```
sum(109,20)
```

Out[3]:

```
129
```

▶ Lambda with Single Argument

Lambda functions can work with a single input.

- Common use cases include squaring, cubing, or transforming values
- Often used in mathematical and data processing tasks

In [4]:

```
# single arguements
# cube
cube = lambda x:x**3
```

In [5]:

```
cube(3)
```

Out[5]:

```
27
```

▶ Lambda with Multiple Arguments

Lambda can accept multiple inputs.

- Useful for calculations like averages, sums, or comparisons
- Keeps logic compact and readable

In [6]:

```
# 3-4
mean_of_3=lambda x,y,z: (x+y+z)/3
```

In [7]:

```
mean_of_3(10,12,236)
```

Out[7]:

```
86.0
```

▶ Lambda Without Arguments

Lambda functions can also be defined without arguments.

- Mostly used for returning fixed values
- Helpful in testing or quick responses

In [8]:

```
# lambda function--? no arguements
greet = lambda :  'Hello world'
greet()
```

Out[8]:

```
'Hello world'
```

▶ Conditional Lambda Functions

Lambda supports **conditional logic** using:

- If–else expressions
- Useful for checks like even/odd or positive/negative
- Makes decision-based logic concise

In [9]:

```
# conditioned based
# even odd
tell = lambda x: 'even' if x%2==0 else 'odd'
```

In [10]:

```
tell(2)
```

Out[10]:

```
'even'
```

In [11]:

```
tell(1)
```

Out[11]:

```
'odd'
```

In [12]:

```
# positive negative
tell2 = lambda x: 'positive' if x>=0 else 'negative'
```

In [13]:

```
tell2(12)
```

Out[13]:

```
'positive'
```

In [14]:

```
tell2(-8)
```

```
Out[14]: 'negative'
```

🔹 Map Function

▶ What is the Map Function?

The **map function** applies a given function to **each element** of an iterable (like a list).

- It processes elements **one by one**
- Returns a **map object (iterator)**
- Output can be converted into a list or other structure

▶ Why Use Map?

- Eliminates the need for loops
- Makes code faster and cleaner
- Ideal for transforming data
- Widely used in data preprocessing

In [15]:

```python
# aam zindagi
l1=[1,2,3,4,5,56,783,23,23]
square= lambda x:x**2
# square(l1)
n1=[]
for i in l1:
    # print(square(i))
    n1.append(square(i))

n1
```

```
Out[15]: [1, 4, 9, 16, 25, 3136, 613089, 529, 529]
```

▶ Map with Lambda Functions

- Lambda functions are commonly used with map
- Each element is passed to the lambda logic
- Output contains transformed values

▶ Real-World Use Cases of Map

- Squaring or cubing numbers
- Converting data formats
- Applying mathematical formulas
- Data cleaning and transformation

In [16]:
```
list(map(square,l1)) #mentos zindagi
```

Out[16]:
```
[1, 4, 9, 16, 25, 3136, 613089, 529, 529]
```

In [17]:
```
l1=[2,5,7,10] #--> cube
list(map(lambda x:x**3,l1))
```

Out[17]:
```
[8, 125, 343, 1000]
```

In [18]:
```
# even odd
list(map(lambda x: 'even' if x%2==0 else 'odd',l1))
```

Out[18]:
```
['even', 'odd', 'odd', 'even']
```

🔹 Anonymous Functions

▶ What are Anonymous Functions?

Anonymous functions are functions **without a name**.

- Lambda functions are anonymous by nature
- Used for quick operations
- Not stored for reuse

They are commonly used **inside map, filter, and reduce**.

In [19]:
```
# anonymous functions

sum(10,20)
```

Out[19]:
```
30
```

In [20]:
```
(lambda x,y:x+y)(10,20)
```

Out[20]:
```
30
```

🔹 Reduce Function

▶ What is Reduce?

The **reduce function** performs a **cumulative operation** on elements of an iterable.

- It combines elements step by step
- Returns a **single final value**
- Works from left to right

▶ How Reduce Works Conceptually

- Takes first two elements and applies operation
- Result is combined with the next element
- Continues until only one value remains

▶ Why Use Reduce?

- Ideal for aggregation tasks
- Used for sum, product, maximum, minimum
- Makes repetitive calculations concise

▶ When to Avoid Reduce

- If logic is complex, normal loops are clearer
- Overuse can reduce readability

---

syntax

reduce(function, iterable)

In [21]:

```
# reduce
from functools import reduce
```

In [22]:

```
# aam zindagi
l1=[1,2,3,4,5,6]
sum=0
for i in l1:
    sum+=i
sum
```

```
Out[22]: 21
```

```
In [23]: # mentos zindagi
         reduce(lambda x,y:x+y,l1)
```

```
Out[23]: 21
```

🔹 Filter Function

▶ What is the Filter Function?

The **filter function** selects elements from an iterable based on a **condition**.

- Keeps elements where condition is `True`
- Removes elements where condition is `False`
- Returns an iterator

▶ Why Use Filter?

- Simplifies conditional selection
- Replaces lengthy loop + if logic
- Makes data filtering efficient

▶ Filter with Lambda

- Lambda defines the condition
- Each element is checked
- Only valid elements are kept

- filter(function, iterable)

```
In [24]: # filter
         l1 = list(range(0,50,3))
         l1
```

```
Out[24]: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

```
In [25]: list(filter(lambda x: True if x>25 else False,l1))
```

```
Out[25]: [27, 30, 33, 36, 39, 42, 45, 48]
```

🔹 Recursion

▶ What is Recursion?

Recursion is a programming technique where a **function calls itself** to solve a problem.

- The problem is broken into smaller parts
- Each call works on a reduced version
- Continues until a base condition is met

▶ Base Case in Recursion

- Base case stops the recursion
- Prevents infinite function calls
- Mandatory for every recursive function

▶ Recursive Case

- Defines how the function calls itself
- Gradually moves toward the base case

▶ Why Use Recursion?

- Useful for problems with repetitive structure
- Ideal for mathematical problems
- Makes logic more intuitive in some cases

▶ Examples of Recursion Use

- Factorial calculation
- Fibonacci series
- Tree and graph traversal
- Divide-and-conquer algorithms

🔹 Recursion vs Loop

- Recursion is elegant but uses more memory

- Loops are faster and more memory efficient
- Choose recursion when problem structure is naturally recursive

---

In [26]:

```
# factorial
# 5! = 5*4*3*2*1
def factorial(n):
    if n==0 or n==1:
        return 1
    else:
        return n*factorial(n-1)
factorial(5)
```

Out[26]:

```
120
```

In [27]:

```
# 1! = 1
# 0! = 1
```

In [28]:

```
n = int(input('enter a number'))
prod = 1
for i in range(1,n+1):
    prod*=i
prod
```

Out[28]:

```
120
```

✅ Final Summary

Advanced functions like **lambda, map, reduce, filter, and recursion** help write:

- Cleaner code
- More efficient logic
- Professional and scalable programs

Mastering these concepts is essential for **data analysts, data scientists, and Python developers**.

In [ ]:

```

```