

# Functions

## 1. What is the difference between a function and a method in Python?

**Function** → A block of reusable code defined with **def** or **lambda**. It stands alone and can be called independently.

For example,

```
[1] ✓ 0s def multiply(x, y):  
        return x*y  
  
[2] ✓ 0s print(multiply(5,9))  
⇒ 45  
  
[3] ✓ 0s multiply(8,3)  
⇒ 24
```

**Method** → A function that is associated with an **object** (part of a class). It's called using object.method(), and usually operates on that object's data.

For example,

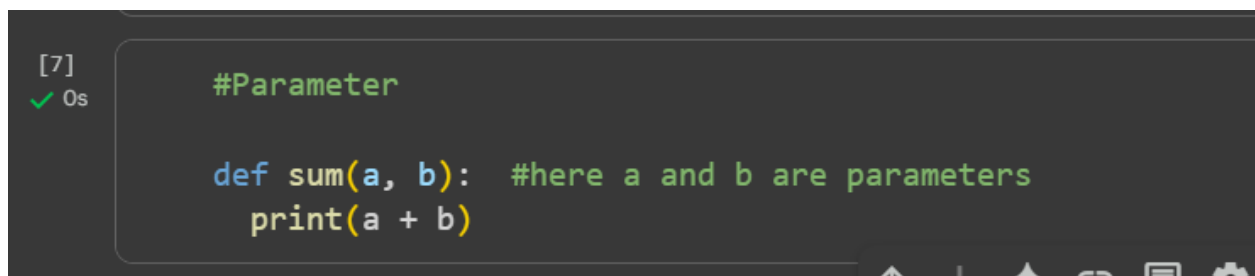
```
[5] ✓ 0s class Math:  
        def div(self, a, b):  
            return a//b  
  
        obj = Math()  
        print(obj.div(25, 5))  
⇒ 5
```

## 2. Explain the concept of function arguments and parameters in Python.

### Function Parameters:

Parameters are the placeholders specified inside the parentheses of a function definition. Each parameter represents a variable that the function intends to work with.

For example,

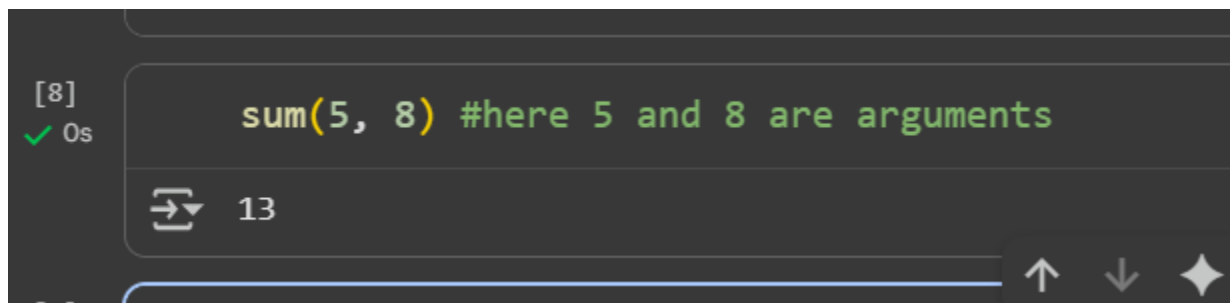


A screenshot of a Jupyter Notebook cell. On the left, it shows the cell index [7], a green checkmark, and the execution time 0s. The code inside the cell is: `#Parameter`, `def sum(a, b): #here a and b are parameters`, and `print(a + b)`. The code is color-coded: `def` is blue, `sum` is green, `(a, b)` is yellow, `:` is blue, `#here a and b are parameters` is green, `print` is blue, `(a + b)` is yellow, and `)` is blue.

### Function Arguments:

Arguments are the actual values passed to the function when it is called. They fill in the parameters defined in the function, providing the specific data the function will use.

For example,



A screenshot of a Jupyter Notebook cell. On the left, it shows the cell index [8], a green checkmark, and the execution time 0s. The code inside the cell is: `sum(5, 8) #here 5 and 8 are arguments`. The code is color-coded: `sum` is green, `(5, 8)` is yellow, `:` is blue, `#here 5 and 8 are arguments` is green. Below the code, the output is shown: a blue icon followed by the number 13. At the bottom right of the cell, there are three icons: an up arrow, a down arrow, and a star.

## Types of Arguments:

**1. Positional Arguments:** It's mapped to parameters by position/order. For example,

```
[9]
✓ 0s
def add(x, y):
    return x + y

print(add(10, 20)) #here 10 is assigned to x, and
                  # 20 is assigned to y based on their positions.
30
```

**2. Keyword Arguments:** Explicitly assign values to parameters using param=value syntax, so order doesn't matter. For example,

```
[15]
✓ 0s
def st_info(name, grade):
    print(f"Student name is {name}, studing in class {grade}.")

[16]
✓ 0s
st_info(name= "Kirti", grade= 9)
st_info(name= "Yash", grade= 10)

Student name is Kirti, studing in class 9.
Student name is Yash, studing in class 10.
```

**3. Default Arguments:** Parameters have default values, so arguments can be omitted during the call.

```
[30]
✓ 0s

def people(name, age='not mention'):
    print(f'Person name is {name} and age is {age}.')

people(name="Kirt", age= 21)
people(name="yashi")

Person name is Kirt and age is 21.
Person name is yashi and age is not mention.
```

#### 4. Arbitrary Positional Arguments (\*args):

Allows passing a variable number of positional arguments.

For example,

```
[35]
✓ 0s

def fruits(*fruit):
    return fruit

#All positional arguments are packed into a tuple called numbers.


print(fruits('apple', 'banana', 'kiwi'))
print(fruits('mango', 'lemon'))


('apple', 'banana', 'kiwi')
('mango', 'lemon')
```


## 5. Arbitrary Keywords Arguments (\*kwargs):

Allows passing a variable number of keyword arguments.

For example,

```
[ ]  def print_info(**info):  
    for key, value in info.items():  
        print(f"{key}: {value}")  
  
    print_info(name= "Kirti", age= 21, city= "varanasi", course= "DS")
```

```
 name: Kirti  
age: 21  
city: varanasi  
course: DS
```

[ ]  Start coding or generate with AI

↑ ↓ ✦ 🔗 💬 ⚙️ 📄 🗑️ ⋮

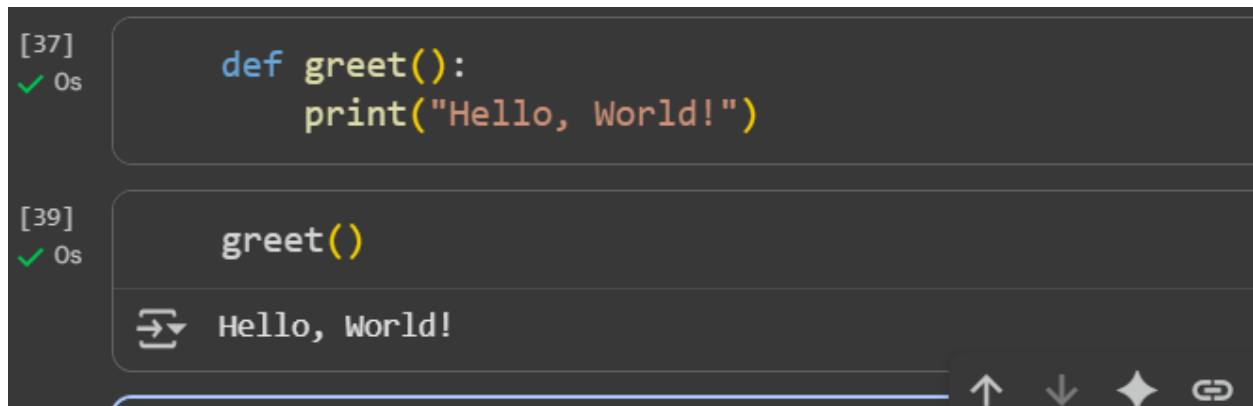
### 3. What are the different ways to define and call a function in Python?

There are several ways to define and call functions in Python. The most common methods are :

#### 1. Standard Function Definition Using def :

Functions are defined using the `def` keyword, followed by the function name, parentheses (which may include parameters), a colon, and an indented block of code (the function body).

For example,



The screenshot shows a code editor with two cells. The first cell, labeled [37], contains the function definition: `def greet():` followed by an indented `print("Hello, World!")`. The second cell, labeled [39], contains the function call `greet()`. Below the call, the output "Hello, World!" is displayed. The interface includes a status bar at the bottom with navigation icons.

```
[37] def greet():  
    print("Hello, World!")  
  
[39] greet()  
  
Hello, World!
```

**2. Function with Parameters:** Functions can accept inputs via parameters, allowing them to operate on different values.



The screenshot shows a code editor with two cells. The first cell, labeled [40], contains the function definition: `def greet(name):` followed by an indented `print(f"Hello, {name}!")`. The second cell, labeled [41], contains the function call `greet('Kirti')`. Below the call, the output "Hello, Kirti!" is displayed. The interface includes a status bar at the bottom with navigation icons and a prompt to "Start coding or generate with AI."

```
[40] def greet(name):  
    print(f"Hello, {name}!")  
  
[41] greet('Kirti')  
  
Hello, Kirti!
```

Start coding or generate with AI.

### 3. Function with Default Parameters:

Parameters can have default values, so callers can omit those arguments.

```
[42] ✓ 0s
def greet(name="Guest"):
    print(f"Hello, {name}!")

[43] ✓ 0s
greet("khushi")
greet()

⇒ Hello, khushi!
   Hello, Guest!
```

### 4. Lambda Functions (Anonymous Functions):

Python supports small anonymous functions with the lambda keyword, typically used for short, simple operations.

```
add = lambda x, y: x + y

print(add(3, 5))

⇒ 8
```

#### 4. What is the purpose of the `return` statement in a Python function?

The purpose of the **return** statement in a Python function is that it provides output (a result ) that can be stored in a variable or used directly in expressions.

For example,

```
def add(a, b):  
    return a+b  
  
result = add(5, 6)  
  
result    # now this result value can be use anywhere  
⇒ 11
```

#### 5. What are iterators in Python and how do they differ from iterables?

An **Iterator** is an object that produces the next value of an iterable on demand. It remembers its current position and moves step by step until elements are exhausted.



For example,

```
my_list = [1, 2, 3]
iterator = iter(my_list)  # turns iterable into iterator

print(next(iterator))

1

print(next(iterator))

2

print(next(iterator))

3

print(next(iterator))

-----
StopIteration                                Traceback (most recent call last)
/tmp/ipython-input-3866658878.py in <cell line: 0>()
----> 1 print(next(iterator))

StopIteration:
```

Next steps: [Explain error](#)

It implements two methods:

1. **iter()** → returns the iterator object itself.
2. **next()** → returns the next element; raises **StopIteration** when no elements are left.

While an **Iterable** is any Python object that can return its elements one by one when asked like list, tuple, string, set, dict, etc.

```
my_list = [1, 2, 3]
for item in my_list:  # works because list is iterable
    print(item)

1
2
3
```

## 6. Explain the concept of generators in Python and how they are defined.

**Generators** are a special kind of iterator in Python. They allow you to produce values one at a time instead of creating and storing the entire sequence in memory. Useful when dealing with large data sets or infinite sequences.

**There are two ways to define Generators:**

### 1. Generator Functions :

These are regular functions but use the `yield` keyword to produce a value and temporarily suspend execution. When resumed, they pick up where they left off.

For example,

```
def count_upto(n):  
    i = 1  
    while i <= n:  
        yield i  
        i += 1
```

```
g = count_upto(3)  
print(next(g))
```

```
⇒ 1
```

```
print(next(g))
```

```
⇒ 2
```

```
print(next(g))
```

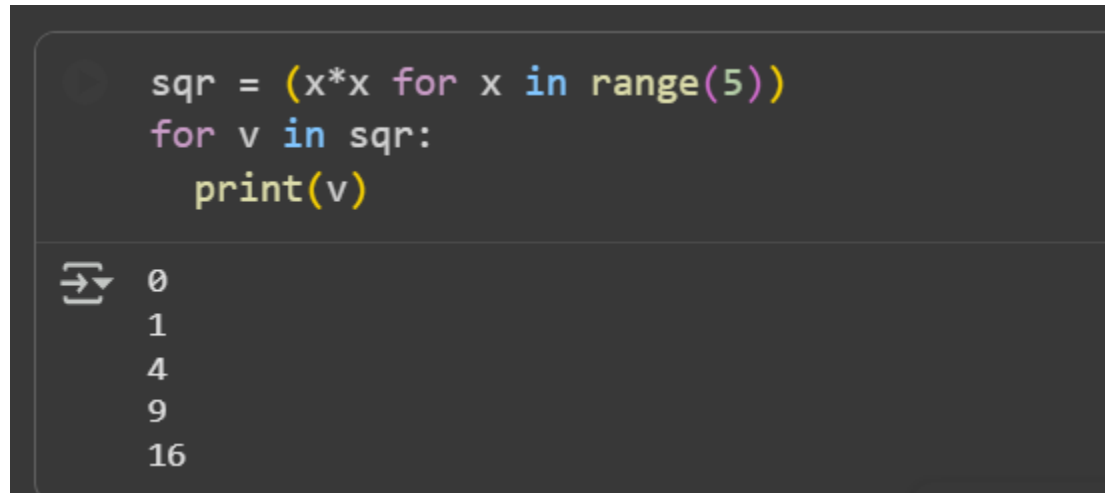
```
⇒ 3
```

## 2. Generator Expressions:

These are similar to list comprehensions but use parentheses () instead of square brackets []. They create generator objects instead of lists.

For example,

```
sqr = (x*x for x in range(5))
for v in sqr:
    print(v)
```

A screenshot of a code editor with a dark background. The code defines a generator expression 'sqr' and iterates over it, printing the values. The output shows the squares of numbers 0 through 4: 0, 1, 4, 9, and 16.

0  
1  
4  
9  
16

## 7. What are the advantages of using generators over regular functions?

The advantages of using generators over regular functions are:

1. Generators don't store the entire result in memory. They produce values one at a time.

For example,

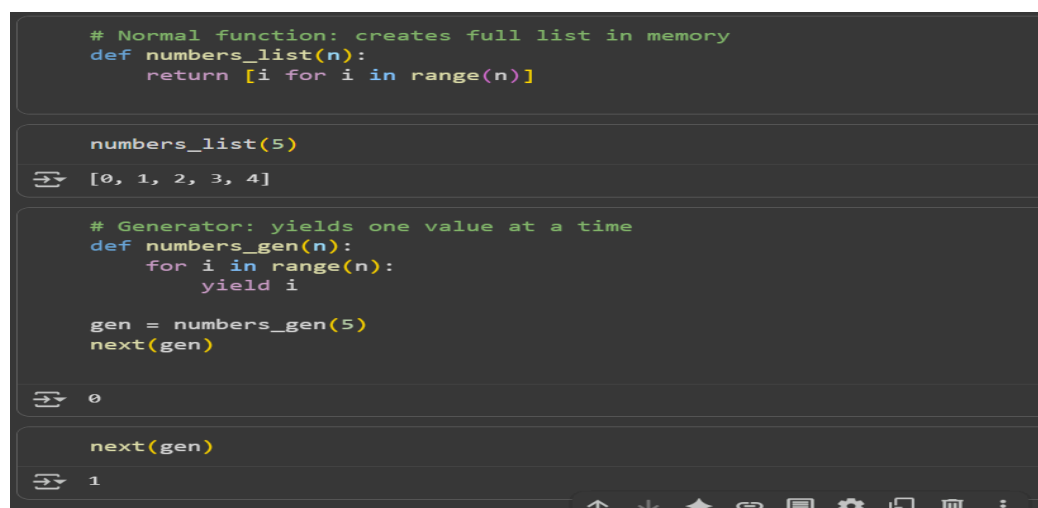
```
# Normal function: creates full list in memory
def numbers_list(n):
    return [i for i in range(n)]

numbers_list(5)
[0, 1, 2, 3, 4]

# Generator: yields one value at a time
def numbers_gen(n):
    for i in range(n):
        yield i

gen = numbers_gen(5)
next(gen)
0

next(gen)
1
```

A screenshot of a code editor showing two functions: 'numbers\_list' which returns a full list, and 'numbers\_gen' which is a generator that yields values one at a time. The output shows the list [0, 1, 2, 3, 4] for the first function, and the values 0 and 1 for the generator as it is iterated.

0  
1

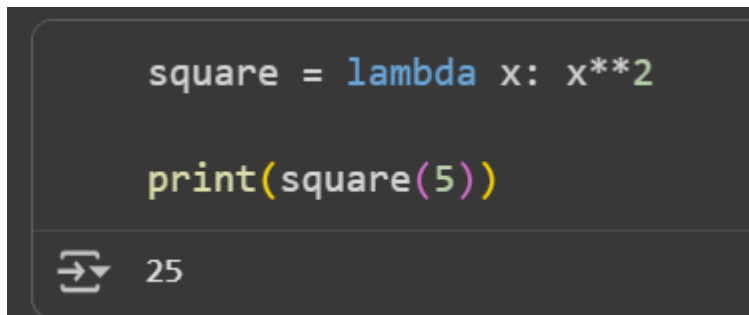
2. Values are computed only when needed, not in advance in the generators.
3. Regular functions can't return infinite sequences while Generators can represent infinite streams because they yield step by step.
4. Faster for streaming data (files, network, logs) since they don't create large intermediate structures.
5. Yield makes iterator logic concise.

## 8. What is a lambda function in Python and when is it typically used?

A **lambda function** in Python is a small, anonymous function defined using the **lambda** keyword instead of **def**.

**Syntax: lambda arguments: expression** → It takes any number of arguments but only one expression, which is evaluated and returned.

```
square = lambda x: x**2  
print(square(5))
```



For example,

Lambda functions are usually used for quick, short-term operations where a full function definition would be unnecessary.

Common use cases:

### 1. Inside built-in functions like `map()`, `filter()`, `reduce()`.

Passing as arguments to higher-order functions like `map()`, `filter()`, and `reduce()` to perform inline operations.

For example,

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, nums))

print(squares)
```

⇒ [1, 4, 9, 16]

### 2. Sorting with a key function:

```
fruits = ["apple", "banana", "kiwi"]
fruits.sort(key = lambda x: len(x))
print(fruits)
```

⇒ ['kiwi', 'apple', 'banana']

### 3. Filtering items:

```
nums = [10, 15, 20, 25, 30]

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens)
```

→ [10, 20, 30]

### 4. When function definition is not worth it:

Lambda are ideal for concise, one-time-use functions in places where defining a regular function would be overkill.

## 9. Explain the purpose and usage of the `map()` function in Python.

The purpose of **map() function** in Python is to apply a given function to every item in an iterable and return a map object.

**Syntax:** map(function, iterable)

The usage are:

#### 1. Using as a Normal Function

```
def square(x):
    return x**2

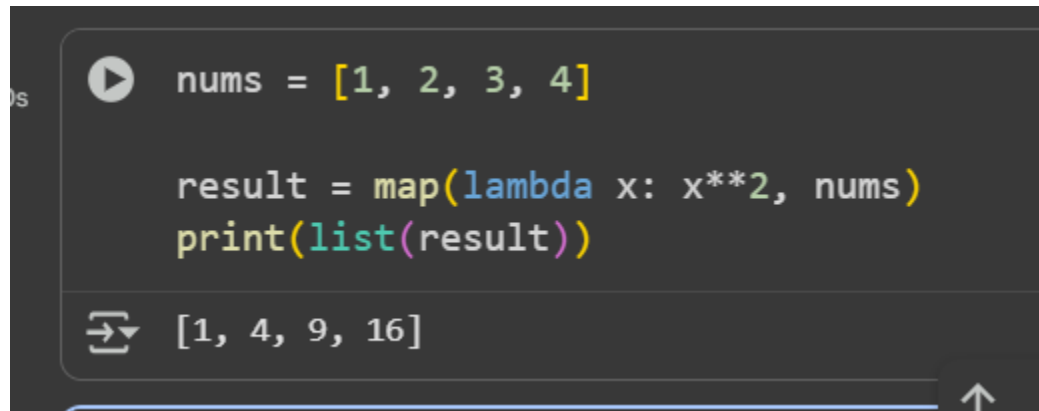
nums = [1, 2, 3, 4]
result = map(square, nums)
print(list(result))
```

→ [1, 4, 9, 16]

For example,

## 2. Using lambda

For example,



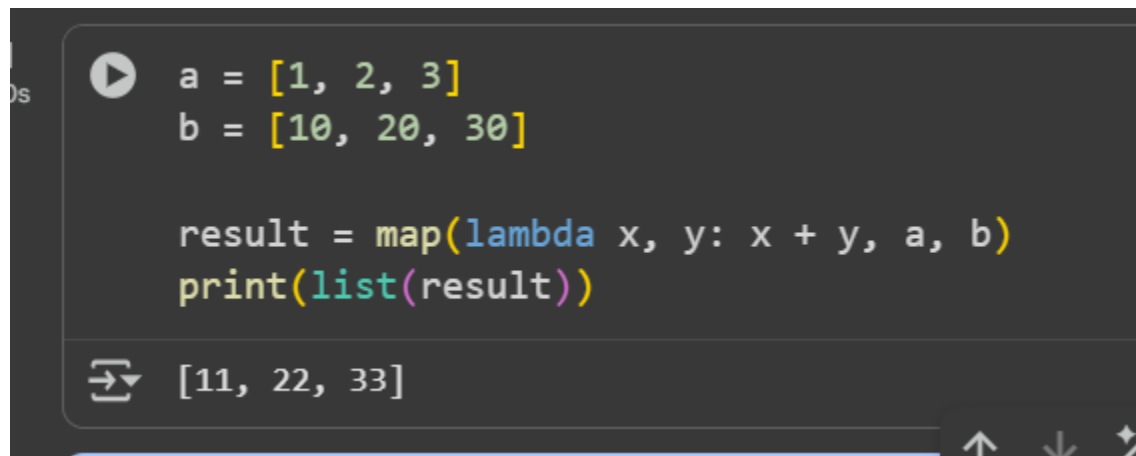
```
nums = [1, 2, 3, 4]

result = map(lambda x: x**2, nums)
print(list(result))
```

[1, 4, 9, 16]

## 3. With Multiple Iterables

If you pass more than one iterable, `map()` applies the function in parallel (like zipping them). For example,



```
a = [1, 2, 3]
b = [10, 20, 30]

result = map(lambda x, y: x + y, a, b)
print(list(result))
```

[11, 22, 33]

## 10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

**map()** : It applies a function to each element of an iterable. As an output a new iterable (map object) with transformed elements.

For example,

```
nums = [1, 2, 3, 4]

squares = list(map(lambda x: x**2, nums))
print(squares)
```

[1, 4, 9, 16]

**filter()** : It selects elements from an iterable based on a condition (True/False). As an iterable (filter object) with only the elements that pass the text.

For example,

```
nums = [1, 2, 3, 4, 5]

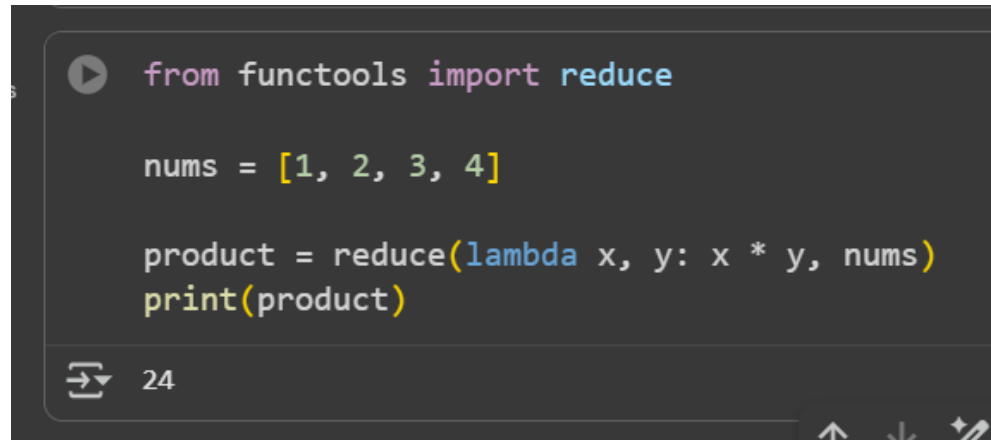
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)
```

[2, 4]



**reduce()** : It repeatedly applies a function to pairs of elements, reducing the iterable to a single value. As a result one final result comes.

For example,

A screenshot of a code editor with a dark background. The code is written in Python and uses the reduce function from the functools module to calculate the product of a list of numbers. The code is as follows:

```
from functools import reduce

nums = [1, 2, 3, 4]

product = reduce(lambda x, y: x * y, nums)
print(product)
```

Below the code, there is a line showing the output of the print statement: `24`. The code is color-coded: `from` is purple, `functools` is green, `import` is purple, `reduce` is blue, `nums` is green, `[1, 2, 3, 4]` is yellow, `product` is green, `lambda` is blue, `x` is green, `y` is green, `:` is green, `x * y` is yellow, `nums` is green, `)` is green, `print` is blue, and `(product)` is green.

11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list: [47, 11, 42, 13].

```
nums = [47, 11, 42, 13]
```

```
result = reduce(lambda x, y: x + y, nums)  
print(result)
```

Internal Mechanism step by step:

The list is [47, 11, 42, 13]

So `reduce(lambda x, y: x + y, nums)` works like:

Step 1: Take first two elements  $\rightarrow 47 \& 11$   
result  $\rightarrow 58$

Step 2: Take the result (58) and the next element (42)  
 $\rightarrow 58 \& 42$   
result  $\rightarrow 100$

Step 3: Take the result (100) and the next element (13)  
 $\rightarrow 100 \& 13$   
result  $\rightarrow 113$

$\therefore$  Final result  $\rightarrow 113$

How reduce function works internally:

$\text{reduce}(\text{func}, [a, b, c, d]) = \text{func}(\text{func}(\text{func}(a, b), c), d)$