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

Functions

- **Definition**: Functions are defined using the `def` keyword and can exist independently of any class.
- **Usage**: They can be called directly by their name.

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

result = add(5, 3)
print(result)  # Output: 8
```

Methods

- **Definition**: Methods are functions that are defined within a class and are associated with an object.
- **Usage**: They are called on an object and can access and modify the object's attributes.

```
class Calculator:
    def add(self, a, b):
        return a + b

calc = Calculator()
result = calc.add(5, 3)
print(result)  # Output: 8
```

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

Parameters

- **Definition**: Parameters are the variables listed inside the parentheses in the function definition.

- **Purpose**: They act as placeholders for the values that will be passed to the function.

```
def greet(name):
   print(f"Hello, {name}!")
```

Arguments

- **Definition**: Arguments are the actual values passed to the function when it is called.
- **Purpose**: They provide the data that the function will use to perform its operations.

```
greet("Alice")
```

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

Defining Functions

1. Standard Function Definition:

```
def greet(name):

   return f"Hello, {name}!"
```

2. Lambda Functions:

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

print(add(2, 3))  # Output: 5
```

3. Nested Functions:

```
def outer():

   def inner():

      return "Inner function"

   return inner()
```

4. Functions with Default Arguments:

```
def greet(name="Guest"):
   return f"Hello, {name}!"
```

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

1. Exiting the Function

- The `return` statement immediately terminates the function's execution and returns control to the caller.

```
def greet(name):
   return f"Hello, {name}!"
   print("This will not be printed")  # This line will not execute
```

2. Returning a Value

- It allows the function to send a value back to the caller, which can then be used or stored.

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


result = add(5, 3)
print(result)  # Output: 8
```

3. Returning Multiple Values

- Python functions can return multiple values as a tuple.

```
def get_coordinates():
    return 10, 20


x, y = get_coordinates()
print(x, y)  # Output: 10 20
```

4. Returning None

- If no `return` statement is used, or if `return` is used without an expression, the function returns None by default.

```
def do_nothing():
    pass


result = do_nothing()
print(result)  # Output: None
```

5. Conditional Returns

- The `return` statement can be used conditionally to exit the function based on certain conditions

```
def check_even(number):
    if number % 2 == 0:
        return True
    return False


print(check_even(4))  # Output: True
print(check_even(5))  # Output: False
```

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

Iterables

- **Definition**: An iterable is any Python object capable of returning its members one at a time, allowing it to be looped over in a `for` loop.

```
my_list = [1, 2, 3]
for item in my_list:
    print(item)
```

Iterators

- **Definition**: An iterator is an object that represents a stream of data. It returns the next item in the sequence when you call the `__next__()` method.

```
my_list = [1, 2, 3]
iterator = iter(my_list)
print(next(iterator))  # Output: 1
print(next(iterator))  # Output: 2
print(next(iterator))  # Output: 3
```

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

Generators in Python are a special type of function that allow you to create iterators in a more memory-efficient way. They generate values on-the-fly and yield them one at a time, rather than returning them all at once. This makes them particularly useful for working with large datasets or streams of data.

Defining Generators

Generators are defined using the `def` keyword, just like regular functions, but they use the `yield` statement instead of `return`. When a generator function is called, it returns a generator object without executing the function immediately. The function's code runs only when the generator's `__next__()` method is called.

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
print(next(counter))  # Output: 1
print(next(counter))  # Output: 2
print(next(counter))  # Output: 3
```

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

1. Memory Efficiency

```
def generate_numbers(n):
    for i in range(n):
        yield i

gen = generate_numbers(1000000)
```

2. Improved Performance

```
def square_numbers(nums):
    for num in nums:
        yield num * num

squares = square_numbers(range(1000000))
```

3. Simplified Code

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

for num in fibonacci(100):
    print(num)
```

4. Infinite Sequences

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

```
for i in infinite_sequence():
    if i > 10:
        break
    print(i)
```

5. Enhanced Iteration Control

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

cd = countdown(5)
print(next(cd))  # Output: 5
print(next(cd))  # Output: 4
```

6. Reduced Complexity

```
def read_lines(file_path):
    with open(file_path) as file:
        for line in file:
            yield line.strip()

for line in read_lines("example.txt"):
    print(line)
```

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. Unlike regular functions created with the `def` keyword, lambda functions are typically used for short, simple operations and are defined in a single line.

Typical Uses of Lambda Functions

1. Short, Simple Functions: When you need a small function for a short period of time, especially within another function.

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

squares = list(map(lambda x: x * x, numbers))

print(squares)  # Output: [1, 4, 9, 16, 25]
```

2. Higher-Order Functions: Often used with functions like `map()`, `filter()`, and `reduce()`.

- **Map**: Applies a function to all items in an input list.

```
doubled = list(map(lambda x: x * 2, numbers))

print(doubled)  # Output: [2, 4, 6, 8, 10]
```

3. Sorting and Key Functions: Used as a key function in sorting algorithms.

```
points = [(1, 2), (3, 1), (5, -1)]

points_sorted = sorted(points, key=lambda point: point[1])

print(points_sorted)  # Output: [(5, -1), (3, 1), (1, 2)]
```

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

Purpose of `map()`

- **Transformation**: It applies a function to all items in an input iterable, transforming them into a new iterable.
- **Functional Programming**: Supports a functional programming style by allowing you to process data without explicit loops.

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


numbers = [1, 2, 3, 4]
result = map(square, numbers)
print(list(result))  # Output: [1, 4, 9, 16]
```

Q10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

1. `map()` Function

- **Purpose**: Applies a given function to each item of an iterable (like a list) and returns a map object (which is an iterator) with the results.

  ```
  numbers = [1, 2, 3, 4]
  result = map(lambda x: x * 2, numbers)
  print(list(result))  # Output: [2, 4, 6, 8]
  ```

2. `filter()` Function

- **Purpose**: Filters elements from an iterable based on a function that returns `True` or `False`.

```
numbers = [1, 2, 3, 4, 5, 6]
```

```
result = filter(lambda x: x % 2 == 0, numbers)

print(list(result))  # Output: [2, 4, 6]
```

3. reduce() Function

- **Purpose**: Applies a function cumulatively to the items of an iterable, reducing the iterable to a single value.

  ```
  from functools import reduce

  numbers = [1, 2, 3, 4]

  result = reduce(lambda x, y: x * y, numbers)

  print(result)  # Output: 24
  ```

  Q11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list: [47,11,42,13]

  ```
  from functools import reduce


  numbers = [47, 11, 42, 13]

  sum_result = reduce(lambda x, y: x + y, numbers)

  print(sum_result)  # Output: 113
  ```

Page No.

Date.

Q11 using pen and paper write the internal mechanism for sum operation using Reduce function on this given list :

list : [47, 11, 42, 13]

```
From func import reduce

numbers = [ 47, 11 , 42, 13]

sum_result = reduce (lambda. x, y : x+y, num)

print (sum_result)
```

<u>output</u> : 113