Functions

Q1. What is the difference between a function and a method in Python?
Ans: functions and methods are both blocks of code that perform specific tasks, but they differ in their association with objects and classes:

Functions

- **Independent:** Functions are standalone entities that can be called from anywhere in your code.
- **No Object Association:** They don't belong to any specific object or class.
- **Direct Invocation:** You call a function directly by its name, followed by parentheses and any necessary arguments.

Example:

```
def greet(name):
    print("Hello,", name + "!")
greet("Alice") # Output: Hello, Alice!
```

Methods

- **Object-Oriented:** Methods are associated with objects or classes.
- **Class Membership:** They are defined within a class.
- **Object Invocation:** You call a method on an object using dot notation (`.`) after the object's name.

Example:

```
class Car:
    def start(self):
        print("The car is starting.")
my_car = Car()
my_car.start() # Output: The car is starting.
```

Q2. Explain the concept of function arguments and parameters in Python.
Ans: **Parameters**

- **Placeholders:** Parameters are the variables defined within the parentheses of a function's definition. They act as placeholders for the values that will be passed to the function when it's called.

Example:

Python
```
def greet(name):  # 'name' is a parameter
    print("Hello,", name + "!")
```

Arguments

- **Actual Values:** Arguments are the actual values that you provide to a function when you call it. These values are passed to the function and are assigned to the corresponding parameters.

Example:

Python
```
greet("Alice")  # "Alice" is an argument
```

Q3. What are the different ways to define and call a function in Python?
Ans: Defining a Function:
Syntax:

Python
```
def function_name(parameter1, parameter2, ...):
    """Docstring: Briefly explain what the function does."""
    # Function body - code to be executed
    return value  # Optional: Return a value
```

- **Key Components:**
    - `def`: Keyword to indicate the start of a function definition.
    - `function_name`: A unique identifier for the function (e.g., `calculate_area`, `greet`).
    - `parameters`: Variables that receive the input values when the function is called. These are optional.
    - `docstring`: A string within triple quotes (""" ... """) that describes the function's purpose. Highly recommended for documentation and readability.
    - `function body`: The code block that contains the instructions to be executed when the function is called.
    - `return statement`: (Optional) Specifies the value that the function should output. If no `return` statement is present, the function returns `None` by default.

Calling a Function:

Syntax:

Python
function_name(argument1, argument2, ...)

-
- **Key Concepts:**
    - `function_name`: The name of the function you want to execute.
    - `arguments`: The values that are passed to the function when it's called. They correspond to the parameters defined in the function's definition.

Different Ways to Call a Function:

- **Positional Arguments:** Arguments are passed in the same order as their corresponding parameters in the function definition.
- **Keyword Arguments:** Arguments are passed by their parameter names, allowing for flexibility in the order.
- **Default Arguments:** Parameters have default values assigned within the function definition. If no argument is provided for a parameter with a default value, the default value is used.
- **Variable-Length Arguments:**
    - `*args`: Allows you to pass an arbitrary number of positional arguments as a tuple.
    - `**kwargs`: Allows you to pass an arbitrary number of keyword arguments as a dictionary.

Q4. What is the purpose of the `return` statement in a Python function?

Ans:A **return statement** is used to end the execution of the function call and it “returns” the value of the expression following the return keyword to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A **return statement** is overall used to invoke a function so that the passed statements can be executed.

Example:

```
def fun(n):

   return [n**2, n**3]

res = fun(3)

print(res) #Output [9, 27]
```

Q5. What are iterators in Python and how do they differ from iterables?

Ans: iterable is an object that one can iterate over. It generates an Iterator when passed to iter() method. An iterator is an object, which is used to iterate over an iterable object using the __next__() method. Iterators have the __next__() method, which returns the next item of the object.

For example, a list is iterable but a list is not an iterator. An iterator can be created from an iterable by using the function iter(). To make this possible, the class of an object needs either a method __iter__, which returns an iterator, or a __getitem__ method with sequential indexes starting with 0.

Q6. Explain the concept of generators in Python and how they are defined.

A generator function is a special type of function that returns an iterator object. Instead of using return to send back a single value, generator functions use yield to produce a series of results over time. This allows the function to generate values and pause its execution after each yield, maintaining its state between iterations.

Example:

```
def fun(max):
    cnt = 1
    while cnt <= max:
        yield cnt
        cnt += 1
ctr = fun(5)
for n in ctr:
print(n)
```

Q7. What are the advantages of using generators over regular functions?

Ans: Generators have several advantages over regular functions, including:

- **Memory efficiency:** Generators are more memory efficient than regular functions because they only compute values when needed, rather than storing all values in memory at once. This is especially useful when working with large data sets.
- **Lazy evaluation:** Generators can be used to lazy load data, meaning that data is not loaded until it is needed.
- **Infinite sequences:** Generators can represent infinite streams of data without consuming excessive memory.
- **Pipelining:** Generators enable a series of operations to be chained together efficiently.
- **Readability and maintainability:** Generator functions encapsulate complex iteration logic, enhancing code clarity.
- **Speed:** Generators can be faster than regular loops, especially when working with large data sets.
- **Pause and resume:** A generator function can be paused and resumed multiple times, while a standard function starts, runs, and returns when the function is finished.

Q8. What is a lambda function in Python and when is it typically used?

Ans: **Python Lambda Functions** are anonymous functions means that the function is without a name. As we already know the def keyword is used to define a normal function in Python. Similarly, the lambda keyword is used to define an anonymous function in Python.

In the **example**, we defined a lambda function(upper) to convert a string to its upper case using upper().

s1 = “pwskills”

s2 = lambda func: func.upper()

print(s2(s1))

Q9. Explain the purpose and usage of the `map()` function in Python.

Ans: The `map()` function in Python is used to apply a given function to each item of an iterable (such as a list, tuple, or string) and returns an iterator that yields the results.

Purpose:

- **Transforming Data:** The primary purpose of `map()` is to transform a sequence of values by applying a specific function to each element.
- **Conciseness:** It provides a concise and elegant way to perform operations on collections of data.
- **Efficiency:** Often more efficient than using a traditional loop for simple transformations.

Usage:

- **Syntax:**

  Python
  map(function, iterable1, iterable2, ...)

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

Ans:

Function	Purpose	Returns	Example
map()	Applies a given function to each item of an iterable.	An iterator of the results	`map(lambda x: x*2, [1, 2, 3])` returns an iterator of `[2, 4, 6]`
filter()	Filters elements from an iterable based on a given condition or function.:[47,11,42,13];	An iterator of the filtered elements.	`filter(lambda x: x%2==0, [1, 2, 3, 4, 5])` returns an iterator of `[2, 4]`
reduce()	Aggregates the elements of an iterable to a single cumulative value using a given function.	A single value	`reduce(lambda x, y: x+y, [1, 2, 3])` returns `6` (1+2+3)

Q11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list

Ans: