Functions Basics Assignment (Theory)

Q1. What is the difference between a function and a method in python?

Ans- In Python, a **function** is a standalone block of code that performs a specific task and can be called independently, while a **method** is a function that is associated with an object and is called on that object.

Example:

- Function:

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

- Method:

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


calc = Calculator()
result = calc.add(2, 3)  # Calling the method on an object
```

Here, `add()` is a function, while `add()` within the `Calculator` class is a method.

Q2. Explain the concept of function arguments and parameters in python.

In Python, function arguments are the values you pass to a function when you call it, while parameters are the variables defined in the function's declaration that accept those arguments.

For example:

python

Copy

```
def greet(name):  # 'name' is a parameter
    print(f"Hello, {name}!")


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

In this example, name is the parameter of the greet function, and "Alice" is the argument passed when the function is called.

Q3. What are the different ways to define and call a function in python?

Ans- In Python, you can define a function using the `def` keyword. You can call a function using its name followed by parentheses, optionally including arguments.

There are also two other ways to define functions: using `lambda` for anonymous functions and defining methods within classes.

### Example of a standard function:

```python
def add(x, y):
    return x + y


result = add(2, 3)  # Calling the function
print(result)  # Output: 5
```

### Example of a lambda function:

```python
```

```
multiply = lambda a, b: a * b
print(multiply(4, 5))  # Output: 20
```
This demonstrates standard function definition and calling, as well as using a lambda function.

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

Ans- The `return` statement in a Python function is used to send back a value to the caller of the function, effectively terminating the function's execution. It allows you to utilize the result of the function elsewhere in your code.

For example:

```python
def add(a, b):
    return a + b  # Returns the sum of a and b


result = add(3, 5)  # result will be 8
```

In this example, the function `add` returns the sum of `a` and `b`, which can then be stored or used as needed.

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

Ans- In Python, **iterators** are objects that implement the iterator protocol, which consists of the `__iter__()` and `__next__()` methods. They allow for traversing through a collection of data one element at a time, maintaining state between calls.

Iterables, on the other hand, are objects that can be looped over, such as lists, tuples, or dictionaries. They implement the `__iter__()` method but do not necessarily implement `__next__()`.

Example:

- An iterable: a list, like `my_list = [1, 2, 3]`.

- An iterator from that iterable: `my_iter = iter(my_list)`.

Using the iterator:

```python
print(next(my_iter))  # Output: 1
print(next(my_iter))  # Output: 2
```

In summary, all iterators are iterables, but not all iterables are iterators. Iterators maintain their own state and can be exhausted, while iterables can be traversed multiple times.

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

Ans- Generators in Python are a type of iterable that allows you to iterate through a sequence of values without storing them all in memory at once. They are defined using functions with the `yield` statement instead of `return`. When a generator function is called, it returns a generator object but does not start execution immediately. Each time the generator's `__next__()` method is called (or when iterating over it), execution resumes until it hits the next `yield` statement, which sends a value back to the caller.

### Example

Here's a simple generator that yields the first n square numbers:

```python
def square_numbers(n):
    for i in range(n):
        yield i ** 2

# Usage:
squares = square_numbers(5)
for square in squares:
    print(square)
```

This will output:

```
0

1

4

9

16
```
In this example, `square_numbers` generates square numbers on-the-fly, making it memory efficient.

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

Ans- Generators have several advantages over regular functions:

1. **Memory Efficiency**: Generators yield items one at a time, which allows them to handle large datasets without loading everything into memory at once.

2. **Lazy Evaluation**: They produce values on-the-fly, enabling the consumption of values as needed, which can improve performance.

3. **State Retention**: Generators maintain their state between calls, making it easier to manage complex iterations.

Example: A generator function can be used to generate a series of Fibonacci numbers:

```python
def fibonacci_generator():
   a, b = 0, 1
   while True:
      yield a
      a, b = b, a + b

# Usage
fib = fibonacci_generator()
for _ in range(5):
   print(next(fib))  # Outputs: 0, 1, 1, 2, 3
```

In this example, the generator produces Fibonacci numbers only as needed without pre-computing or storing the entire sequence in memory.

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

Ans- A lambda function in Python is a small anonymous function defined with the `lambda` keyword. It can take any number of arguments but can only have one expression. Lambda functions are typically used for short, throwaway functions, often in situations where a full function definition would be cumbersome, such as with functions like `map()`, `filter()`, or `sorted()`.

Example:

```python
# Using lambda to create a function that squares a number
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

In this example, the lambda function takes an argument `x` and returns its square.

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

Ans- The `map()` function in Python applies a specified function to each item of an iterable (like a list or a tuple) and returns a map object (an iterator). It's commonly used for transforming data without the need for explicit loops.

### Usage:

```python
result = map(function, iterable)
```

- **Purpose**: To efficiently apply a transformation or operation across all items in an iterable.
- **Example**:
  ```python
  numbers = [1, 2, 3, 4]
  ```

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

  # squares will be [1, 4, 9, 16]

In this example, `map()` squares each number in the `numbers` list.

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

Ans- The `map()`, `reduce()`, and `filter()` functions in Python are all higher-order functions used in functional programming, but they serve different purposes:

### `map()`

- **Purpose**: Applies a specified function to every item in an iterable, returning a new iterable with the results.

- **Usage**: `map(function, iterable)`

- **Example**:

  ```python

  numbers = [1, 2, 3, 4]

  squares = list(map(lambda x: x ** 2, numbers))  # Returns [1, 4, 9, 16]

  ```

### `reduce()`

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

- **Usage**: `reduce(function, iterable[, initializer])`

- **Example**:

  ```python

  from functools import reduce

  numbers = [1, 2, 3, 4]

  product = reduce(lambda x, y: x * y, numbers)  # Returns 24 (1*2*3*4)

  ```

### `filter()`

- **Purpose**: Filters items in an iterable based on a boolean function, returning only those for which the function returns `True`.

- **Usage**: `filter(function, iterable)`

- **Example**:

  ```python
  numbers = [1, 2, 3, 4]
  evens = list(filter(lambda x: x % 2 == 0, numbers))  # Returns [2, 4]
  ```

### Summary:

- **`map()`** transforms each item in an iterable.

- **`reduce()`** combines items into a single value.

- **`filter()`** selects items based on a condition.

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

list:[47,11,42,13];
(Attach paper image for this answer) in doc or colab notebook.

Internal Mechanism of Sum Using reduce().

1. Import reduce from functools.
   ⇒ from functools import reduce
2. Define the List.
   ⇒ numbers = [47, 11, 42, 13]
3. Create a Lamba function or a normal function to add two number.
   ⇒ add = Lambda x, y : x+y
4. Reduce operation - Apply reduce() to the list.
   Step 1 - Apply add to the first two elements:
   - add (47, 11) → Result : 58

   Step 2 - Apply add to the result and the next element.
   - add (58, 42) → Result : 100.

   Step 3 - Apply add to the result and the last element
   - add (100, 13) → Result : 113
5. Final Result - reduce() is 113.

Code Representation :-

```
from functools import reduce
numbers = [47, 11, 42, 13]
total = reduce (lambda x, y : x+y, numbers)
print (total).
  # output : 113.
```