ABHISHEK SAHANI

Assignment_3

1.What is the difference between a function and a method in Python? And also give example

=In Python, a function and a method are both blocks of code that can be executed multiple times from different parts of a program. However, there are key differences between them:

Function:

- A function is a self-contained block of code that takes arguments and returns a value.
- It is not associated with any particular class or object.
- It can be called independently, without the need for an object.

Method:

- A method is a function that is bound to a class or object.
- It takes the object as the first argument (usually referred to as self).
- It can access and modify the object's attributes.

Example:

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

greet("Alice")  # Output: Hello, Alice!

# Method
class Person:
    def __init__(self, name):
```

```
        self.name = name

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

alice = Person("Alice")
alice.greet()  # Output: Hello, Alice!
```

In this example, greet is a function that takes a name as an argument and prints a greeting message. The greet method, on the other hand, is bound to the Person class and takes the object (self) as an argument. It accesses the object's name attribute and prints a greeting message.

2.Explain the concept of function arguments and parameters in Python? And also give an example

= In Python, functions can take arguments, which are values passed to the function when it's called. These arguments are assigned to parameters, which are variables that receive the argument values within the function.

Here's a breakdown:

Arguments:

- Values passed to a function when it's called
- Can be constants, variables, or expressions
- Are passed to the function by position or keyword

Parameters:

- Variables that receive the argument values within the function
- Are defined in the function definition
- Can have default values, making them optional

Example:

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

# Positional arguments
greet("Alice", "Hi")  # Output: Hi, Alice!

# Keyword arguments
greet(name="Bob", message="Goodbye")  # Output: Goodbye, Bob!

# Using default parameter value
greet("Charlie")  # Output: Hello, Charlie!
```

In this example:

- name and message are parameters defined in the greet function.
- When calling greet, we pass arguments "Alice" and "Hi", which are assigned to name and message parameters, respectively.
- In the second call, we use keyword arguments to specify the values for name and message.
- In the third call, we omit the message argument, so it uses the default value "Hello".

3. What are the different ways to define and call a function in Python? And also give an example

=In Python, there are several ways to define and call functions. Here are some examples:

Defining Functions

1. Standard Function Definition

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

greet("Alice")
```

1. Lambda Functions (anonymous functions)

```
greet = lambda name: print(f"Hello, {name}!")
greet("Alice")
```

1. Nested Functions (functions inside functions)

```
def outer():
    def inner(name):
        print(f"Hello, {name}!")
    inner("Alice")

outer()
```

Calling Functions

1. Positional Arguments

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

greet("Alice", "Hi")
```

1. Keyword Arguments

```
greet(name="Alice", message="Hi")
```

1. Default Argument Values

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

greet("Alice")
```

1. Variable Argument Lists (*args and **kwargs)

```
def greet(*names, message="Hello"):
    for name in names:
        print(f"{message}, {name}!")
```

```
greet("Alice", "Bob", "Charlie")
```

1. *Function Call with * and ** Operators*

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

args = ["Alice", "Hi"]
kwargs = {"name": "Bob", "message": "Hello"}

greet(*args)
greet(**kwargs)
```

These examples demonstrate various ways to define and call functions in Python, including standard function definitions, lambda functions, nested functions, positional and keyword arguments, default argument values, variable argument lists, and function calls with * and ** operators.

4. What is the purpose of the return statement in a Python function? And also give an example

= The return statement in a Python function serves several purposes:

1. Exiting the function: The return statement terminates the function's execution and passes control back to the caller.

2. Returning a value: The return statement can pass a value back to the caller, which can be stored in a variable, used in an expression, or passed to another function.

3. Specifying a default return value: If a function doesn't explicitly return a value, it implicitly returns None. The return statement can specify a different default return value.

Example:

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

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

In this example:

- The add function takes two arguments, a and b.
- The function calculates the sum of a and b and stores it in the result variable.
- The return statement passes the result value back to the caller.
- The caller stores the returned value in the sum variable and prints it.

Without the return statement, the function would implicitly return None, and the sum variable would be assigned None instead of the calculated sum.

5. What are iterators in Python and how do they differ from iterables? And also give an example

= In Python, iterators and iterables are two related but distinct concepts:

Iterables:

- An iterable is an object that can be iterated over, meaning it can be used in a loop or converted into a sequence (like a list or tuple).
- Examples of iterables include lists, tuples, dictionaries, sets, and strings.
- Iterables have a __iter__() method that returns an iterator object.

Iterators:

- An iterator is an object that keeps track of its position in an iterable and yields values one at a time.
- Iterators have a __next__() method that returns the next value in the sequence.
- Iterators are created by calling the __iter__() method on an iterable.

Key differences:

- An iterable can be iterated over multiple times, while an iterator can only be iterated over once.
- An iterable can be converted into a sequence, while an iterator cannot.

Example:

```
# Iterable (list)
my_list = [1, 2, 3]

# Create an iterator from the iterable
my_iter = iter(my_list)

# Use the iterator to get values
print(next(my_iter))  # prints 1
print(next(my_iter))  # prints 2
print(next(my_iter))  # prints 3

# Try to get another value (will raise StopIteration)
try:
   print(next(my_iter))
except StopIteration:
   print("Iterator exhausted")
# Create a new iterator from the same iterable
my_iter2 = iter(my_list)
```

```
print(next(my_iter2))  # prints 1 again
```

In this example, my_list is an iterable, and my_iter is an iterator created from it. The iterator is used to get values from the iterable, and once exhausted, it cannot be used again. A new iterator can be created from the same iterable to start over.

6. Explain the concept of generators in Python and how they are defined? And also give an example

= In Python, a generator is a special type of iterable that generates values on the fly, rather than storing them in memory. Generators are defined using a function with the yield keyword, which produces a value and suspends execution, allowing the function to resume where it left off.

Here's an example:

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

# Create a generator object
gen = infinite_sequence()

# Print the first 5 values
for _ in range(5):
    print(next(gen))  # prints 0, 1, 2, 3, 4
```

In this example:

- The infinite_sequence function is a generator that yields an infinite sequence of numbers.
- The yield keyword produces a value and suspends execution.
- The next() function is used to retrieve the next value from the generator.
- The generator remembers its state, so it can resume where it left off.

Generators are useful for:

- Creating infinite sequences
- Handling large datasets without loading them into memory
- Implementing cooperative multitasking
- Creating iterators that can be used in loops

Note that generators can only be iterated over once. If you try to iterate over a generator again, it will be exhausted and will not produce any values.

7. What are the advantages of using generators over regular functions? And also give an example

= Advantages of using generators over regular functions:

1. Memory Efficiency: Generators use significantly less memory, as they only store the current state, not the entire sequence.

2. Lazy Evaluation: Generators only compute values when needed, reducing computation time and resources.

3. Flexibility: Generators can be used to create complex sequences and iterables.

4. Improved Performance: Generators can improve performance by avoiding unnecessary computations and memory allocations.

Example:

Regular Function

```
def squares(n):
    result = []
    for i in range(n):
        result.append(i ** 2)
    return result

print(squares(5))  # [0, 1, 4, 9, 16]
```

Generator Function

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

print(list(squares_gen(5)))  # [0, 1, 4, 9, 16]
```

In this example, the regular function squares creates a list of squares, storing all values in memory. The generator function squares_gen yields squares on the fly, using significantly less memory.

Generators are particularly useful when working with large datasets, infinite sequences, or complex computations, where memory efficiency and lazy evaluation are crucial.

8. What is a lambda function in Python and when is it typically used? And also give an example

= A lambda function in Python is a small, anonymous function that can take any number of arguments, but can only have one expression. It's typically used when a small, one-time use function is needed, and defining a full-fledged function with the def keyword would be unnecessary.

Lambda functions are often used:

1. As arguments to higher-order functions (functions that take functions as arguments)
2. As event handlers

3. As quick, one-time use functions

Example:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

In this example, the lambda function lambda x: x ** 2 takes a single argument x and returns its square. The map function applies this lambda function to each element in the numbers list, and the resulting squared numbers are collected in the squared_numbers list.

Lambda functions can also be used to create small, one-time use functions for tasks like data processing, filtering, or sorting.

For example:

```
names = ["John", "Alice", "Bob"]
sorted_names = sorted(names, key=lambda x: x.lower())
print(sorted_names)  # Output: ["Alice", "Bob", "John"]
```

9. Explain the purpose and usage of the map() function in Python. And also give an example

= The map() function in Python is a built-in function that applies a given function to each item of an iterable (such as a list, tuple, or string) and returns a list of the results.

Purpose:

- To apply a function to each element of an iterable
- To transform or process data in a concise and readable way

Usage:

- map(function, iterable)

Example:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

In this example:

- The lambda function lambda x: x ** 2 is applied to each element of the numbers list
- The map() function returns a map object, which is converted to a list using the list() function
- The resulting list squared_numbers contains the squared values of the original numbers

Other examples:

- strings = ["hello", "world"]

- upper_strings = list(map(str.upper, strings))
- print(upper_strings) # Output: ["HELLO", "WORLD"]

- numbers = [1, 2, 3, 4, 5]
- double_numbers = list(map(lambda x: x * 2, numbers))
- print(double_numbers) # Output: [2, 4, 6, 8, 10]

Note: In Python 3, map() returns an iterator, so you need to convert it to a list or other iterable if you need to use it multiple times.

10. What is the difference between map(), reduce(), and filter() functions in Python? And also give an example

= map(), reduce(), and filter() are three fundamental functions in Python's Functional Programming paradigm.

Map()

- Applies a function to each element of an iterable.
- Returns a new iterable with the transformed elements.
- Purpose: Transformation.

Example:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # [1, 4, 9, 16, 25]
```

Reduce()

- Applies a function to the first two elements of an iterable, then to the result and the next element, and so on.
- Returns a single value.
- Purpose: Accumulation.

Example (using functools.reduce):

```
import functools
numbers = [1, 2, 3, 4, 5]
sum = functools.reduce(lambda x, y: x + y, numbers)
print(sum)  # 15
```

Filter()

- Applies a predicate function to each element of an iterable.
- Returns a new iterable with only the elements for which the predicate is True.

- Purpose: Selection.

Example:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]
```

Key differences:

- map() transforms elements, reduce() accumulates elements, and filter() selects elements.
- map() and filter() return iterables, while reduce() returns a single value.

These functions promote concise, readable code and are essential tools for functional programming in Python.

11. Using Pen & Paper write the internal Mechanism for sum operating using reduce function on this given list [47, 11, 42, 13]

⇒ The internal mechanism for the sum operating using the reduce function on the given list

(i) Initial list [47, 11, 42, 13]

(ii) Reduce Function x, y : x+y

(iii) Iteration
- x = 47
- y = 11
- result 47 + 11 = 58

(iv) Iteration 2
- x = 58
- y = 42
- result = 58 + 42 = 100

(v) Iteration
- x = 100
- y = 13
- result = 100 + 13 = 113

Final result = 113

Python code Example

```
import functionool

numbers = [47, 11, 42, 13]
sum_result = functools.reduce (lambda x, y: x+y, numbers)
print (sum_result)   # Output: 113
```

Visualization

```
  [47, 11, 42, 13]
(47+ 11) = 58
(58 +42) = 100
(100 + 13) = 113
```

The reduce function applies the lambda function (addition) cummulatively to the item of the list reducing it to a single output value - the sum of all element.