lambda function is a small, anonymous function defined using the `lambda` keyword. It can take any number of arguments but only contains a single expression. Lambda functions are primarily used for short, simple operations and are often passed as arguments to higher-order functions like `map()`, `filter()`, and `sorted()`.

Syntax of a Lambda Function

lambda arguments: expression

- `lambda` The keyword to define a lambda function.
- `arguments` Input values the function takes (similar to the parameters in a normal function).
- `expression`: A single expression whose result will be returned by the lambda function.

Example of a Lambda Function:

Lambda function to add two numbers

add = lambda x, y: x + y

print(add(5, 3)) # Output: 8

Key Features of Lambda Functions:

- **Anonymous:** Lambda functions don't have a name (unless assigned to a variable like in the example above).

- **Single expression:** They can only consist of a single expression and can't contain multiple statements or complex logic.

- **Implicit return:** The result of the expression is automatically returned; there's no need for a `return` statement.

Where Lambda Functions are Useful:

- Lambda functions are typically used in situations where a function is required temporarily, such as when passing a function as an argument to other functions like `map()`, `filter()`, or `sorted()`.

Using Lambda Functions with `map()`

`map()` applies a function to each item in an iterable (like a list).

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

Using Lambda Functions with `filter()`:

`filter()` returns only the elements of an iterable for which a given condition is true.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4, 6]
```

Using Lambda Functions with `sorted()`:

You can provide a lambda function as a `key` to the `sorted()` function to define custom sorting logic.

```
students = [('John', 20), ('Alice', 22), ('Bob', 19)]
sorted_students = sorted(students, key=lambda x: x[1])  # Sort by age
print(sorted_students)  # Output: [('Bob', 19), ('John', 20), ('Alice', 22)]
```

Lambda with Multiple Arguments

```
multiply = lambda a, b: a * b
print(multiply(3, 4))  # Output: 12
```

Lambda in `reduce():

`reduce()` is used to apply a function cumulatively to the items in a list, reducing them to a single value.

```
from functools import reduce

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

```
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

Limitations of Lambda Functions:

Single expression only:

You can't include multiple expressions or statements in a lambda function.

- Limited readability: Overuse of lambda functions in complex scenarios can make code harder to read and debug.
- No name: Since they are anonymous, lambda functions can't be referenced elsewhere unless assigned to a variable.

When to Use Lambda Functions:

- For simple operations that can be written in a single expression.
- When a function is needed for a short duration or as an argument to other functions.
- To avoid the overhead of defining a full function using the `def` keyword for simple logic.

Example of a Regular Function vs. Lambda Function:

Regular Function:

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

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

Lambda Function:

```
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8
```

Both approaches give the same result, but the lambda version is more concise for simple operations.

Lambda functions in Python are concise, anonymous functions that are useful for short, throwaway operations, especially when used with functions like `map()`, `filter()`, and `sorted()`. However, they should be used judiciously, as more complex operations are better suited for regular functions defined using `def`.