DATA SCIENCE COURSE TUTORIAL # 31

3.18.1 Lambda Functions

What is a Lambda Function?

A **lambda function** in Python is a small, anonymous function defined using the `lambda` keyword. It can take any number of arguments but can only have one expression. Lambda functions are often used for short, simple operations that do not require a full function definition.

Syntax:

```
lambda arguments: expression
```

Example:

```
double = lambda x: x * 2
print(double(5))   # Output: 10
```

Why Use Lambda Functions?

- Used for short, temporary functions.
- Helpful for reducing code length.
- Often used with functions like `map()`, `filter()`, and `reduce()`.

Lambda Function with Multiple Arguments

You can pass multiple arguments to a lambda function.

Example:

```
add = lambda a, b: a + b
print(add(3, 7))   # Output: 10
```

---

map(), filter(), and reduce() Functions

map() Function

The **map()** function applies a given function to every item in an iterable (like a list) and returns a new iterable (map object) with the results.

Syntax:

```
map(function, iterable)
```

Example:

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

Explanation:

- `map()` applies the lambda function `x * 2` to every element in the list.
- The output is a new list with all elements doubled.

---

filter() Function

The **filter()** function filters elements of an iterable based on a condition provided in a function. Only elements that return `True` are included in the result.

Syntax:

```
filter(function, iterable)
```

Example:

```
numbers = [10, 15, 20, 25, 30]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)   # [10, 20, 30]
```

Explanation:

- `filter()` checks each number.
- If the condition `x % 2 == 0` is true, the number is added to the result.

---

reduce() Function

The **reduce()** function applies a function cumulatively to items in a sequence, reducing the iterable to a single value. It is part of the `functools` module.

Syntax:

```
from functools import reduce
reduce(function, iterable)
```

Example:

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)   # 24
```

Explanation:

- `reduce()` first multiplies the first two numbers, then the result with the next, and so on.
- The final output is the cumulative result of all operations.

Lambda in Sorting

You can use a lambda function as a key in sorting operations.

Example:

```
students = [("Ali", 22), ("Sara", 20), ("Ahmed", 25)]
students.sort(key=lambda x: x[1])
print(students)   # [('Sara', 20), ('Ali', 22), ('Ahmed', 25)]
```

Lambda vs Regular Function

Feature	Lambda Function	Regular Function
Definition	Single-line anonymous function	Defined using `def` keyword
Name	Has no name (anonymous)	Has a defined name
Multiple Statements	Not allowed	Allowed
Use Case	Short, simple operations	Complex logic or multiple lines

---

Limitations of Lambda Functions

- Cannot contain multiple statements.
- Difficult to debug and document.
- Best suited for short and simple operations.

3.18.2 Recursion in Functions

What is Recursion?

Recursion is a process where a function calls itself to solve smaller parts of a problem. Each recursive call works on a smaller version of the original problem until a base condition is met.

Basic Structure of a Recursive Function

A recursive function has two parts:

1. **Base Case:** The condition that stops recursion.
2. **Recursive Case:** The function calls itself with new arguments.

Example:

```
def countdown(n):
    if n == 0:
        print("Time's up!")
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

Example: Factorial Using Recursion

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))   # Output: 120
```

Explanation:

- `factorial(5)` calls `factorial(4)` and continues until `factorial(1)`.
- Then results multiply backward when the stack unwinds.

Example: Sum of Natural Numbers Using Recursion

```
def recursive_sum(n):
    if n == 0:
        return 0
    else:
        return n + recursive_sum(n - 1)
```

```
print(recursive_sum(5))   # Output: 15
```

Recursion vs Iteration

Feature	Recursion	Iteration
Definition	Function calls itself	Loop executes repeatedly
Termination	Base condition	Loop condition
Memory Usage	High (stack memory used)	Low
Speed	Slower (function calls)	Faster
Best Use Case	Factorial, Fibonacci, divide problems	Repetitive tasks

---

Common Recursive Problems

- Factorial calculation.
- Fibonacci sequence.
- Sum of numbers.
- String reversal.
- Searching and sorting algorithms (like binary search, merge sort).

Recursive Function Limitations

- May cause **stack overflow** if recursion depth is too large.
- Slower than iteration due to repeated calls.
- Always include a base case to prevent infinite recursion.

Controlling Recursion Limit

Python has a default recursion limit to prevent infinite recursion. You can check or modify it using the `sys` module.

Example:

```
import sys
print(sys.getrecursionlimit())     # Check current limit

sys.setrecursionlimit(2000)        # Increase limit
```