UNIT – 3: Python for Data Processing

3.1 Functional Python Programming  
 **3.1.1 Lambda Functions and Their Utility**

What Is Lambda Function ?

**Lambda functions** are anonymous, inline functions defined using the lambda keyword

**Syntax**

lambda arguments: expression

arguments = inputs to the function

expression = single output returned automatically

**Example**

**1 )** square = lambda x: x \*\* 2

print(square(5))

# Output: 25

**2 )** add = lambda a, b: a + b

print(add(3, 5))

# Output: 8

**Difference Between Normal Function and Lambda Function**

| **Feature** | **Normal Function** | **Lambda Function** |
| --- | --- | --- |
| **Definition** | Defined using the def keyword. | Defined using the lambda keyword. |
| **Name** | Has a name (e.g., def add():) | Anonymous (usually used without a name) |
| **Use Case** | Used for complex or reusable logic. | Used for short, one-time operations. |
| **Readability** | Easier to understand, especially for complex logic. | Less readable if overused or complex. |
| **Return Statement** | Requires an explicit return. | Implicitly returns the expression result. |

Normal 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

When to Use Which?

Use **normal functions** when:

* The logic is complex.
* You need to reuse or debug the function.
* You want better readability.

Use **lambda functions** when:

* You need a quick, one-line function (e.g., with map(), filter()).
* You don't need to reuse or name the function.

**3.1.2 Map, Filter, and Reduce Functions for Efficient Data Processing**

These are built-in functions that operate over iterables (like lists) and follow the principles of functional programming:

Syntax

**map(function, iterable)**

function: What to do to each item

iterable: A list, tuple, etc.

Applies a function to every item in an iterable.

Example

**I want to double every number in my list.**

nums = [1, 2, 3]

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

# Output: [1, 4, 9]

When to use map and when to use for loop ?

map() = cleaner, shorter, good for simple transformations  
for loop = more powerful, better for complex tasks

**Example: Multiply numbers by 2**

**Using map**

numbers = [1, 2, 3, 4]

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

print(result)

# Output: [2, 4, 6, 8]

**using for loop**

numbers = [1, 2, 3, 4]

result = []

for x in numbers:

result.append(x \* 2)

print(result)

# Output: [2, 4, 6, 8]

**Why list() is used around map() ?**

If we not use list then ?

result = map(lambda x: x\*\*2, [1, 2, 3])

print(result)

**Output**  
<map object at 0x000001B7A...>

This is **not a list**, just a special object that stores results lazily (to save memory).

**Other Collection Converters Like list()**

| **Function** | **Converts to** | **Description** | **Example** |
| --- | --- | --- | --- |
| **list()** | List | Converts iterable to a list (ordered, mutable) | list(map(...)) → [1, 2, 3] |
| **tuple()** | Tuple | Converts iterable to a tuple (ordered, immutable) | tuple(filter(...)) → (2, 4, 6) |
| **set()** | Set | Converts iterable to a set (unordered, unique elements) | set([1,2,2,3]) → {1, 2, 3} |
| **dict()** | Dictionary | Converts iterable of key-value pairs to a dict | dict([('a', 1), ('b', 2)]) → {'a': 1, 'b': 2} |

**filter(function, iterable)**

function: What to do to each item

iterable: A list, tuple, etc.

Returns items from an iterable for which the function returns True.

Example

**I only want the even numbers from my list.**

nums = [1, 2, 3, 4]

evens = list(filter(lambda x: x % 2 == 0, nums))

# Output: [2, 4]

**reduce(function, iterable)**

The reduce() function is used when you want to combine all elements in a list into a single result — such as a total, product, maximum, or even a custom combination of values.

* you must import reduce() from the **functools** module.
* Takes a **function** and an **iterable** (like a list).
* The **function** must take **two arguments**.
* It applies the function to the **first two elements**, then to the **result and the next element**, and so on…
* Returns a **single final value**.

Example

#import library  
from functools import reduce

# Function that adds two numbers

def add(x, y):

return x + y

numbers = [1, 2, 3, 4, 5]

result = reduce(add, numbers)

print(result)

# Output: 15

**For Explain Its Working:**

1. add(1, 2) → 3
2. add(3, 3) → 6
3. add(6, 4) → 10
4. add(10, 5) → 15

Question : If my Array like this then what happened  
data = [True, "hello", "world"]

**3.1.3 Using These Functions in Preprocessing Tasks**

**Using map() in Preprocessing**

The map() function is used when you want to apply the same operation to every item in a data set model.

* It saves you from writing a loop, and it's especially helpful when processing **large datasets** during **data cleaning or transformation**.
* Makes your code **shorter and more readable**
* Avoids **for-loops** by applying a function to all items
* Helps in **formatting**, **normalizing**, or **converting data types**
* Often used in **data pipelines** and **ETL (Extract, Transform, Load)** steps

Example: Convert strings to integers

data = ['10', '20', '30']

converted = list(map(int, data))

# [10, 20, 30]

Example: Convert temperatures from Celsius to Fahrenheit

temps\_c = [0, 20, 30]

temps\_f = list(map(lambda c: c \* 9/5 + 32, temps\_c))

# [32.0, 68.0, 86.0]

**Using filter() in Preprocessing**

The **filter()** function is used when you want to **remove items from a list or iterable** that do **not meet a specific condition**.

* It’s perfect for **cleaning data** by keeping only the valid or useful entries and ignoring the rest
* Helps **clean up dirty data**
* Removes **missing, invalid, or irrelevant entries**
* Makes downstream processing (e.g., training models) more accurate
* Keeps your code **cleaner** than using for loops

Example

Datasets contain empty fields that you need to remove.

data = ['apple', '', 'banana', '', 'cherry']

cleaned = list(filter(lambda x: x != '', data))

# Output: ['apple', 'banana', 'cherry']

Example

You could also use bool(x) to remove any "falsy" values like '', None, or 0

data = ['apple', '', None, 'banana']

cleaned = list(filter(bool, data))

# Output: ['apple', 'banana']

Example

Keep Only Valid Emails

emails = ['user@example.com', '', 'not-an-email', 'admin@domain.com']

valid\_emails = list(filter(lambda x: '@' in x and '.' in x, emails))

# Output: ['user@example.com', 'admin@domain.com']

**Using reduce() in Preprocessing**

The reduce() function is used when you want to combine all elements in a list into a single result — such as a total, product, maximum, or even a custom combination of values

* Unlike map() and filter(), you must import reduce() from the **functools** module
* To **summarize or aggregate** data
* To **compress a dataset** into a single value
* Makes your code **cleaner** than writing explicit loops
* Very helpful for **calculating totals**, **finding extremes**, or **combining results**

Example Calculate the Sum of Values

from functools import reduce

data = [10, 20, 30]

total = reduce(lambda x, y: x + y, data)

print(total)

# Output: 60

Example Find the Maximum or Minimum

data = [45, 72, 30, 99, 60]

maximum = reduce(lambda x, y: x if x > y else y, data)

print(maximum)

# Output: 99

Combined Example

from functools import reduce

data = ['10', '20', '', '30', 'not-a-number']

# Step 1: Filter out empty and non-numeric strings

cleaned = filter(lambda x: x.isdigit(), data)

# Step 2: Convert to integers

numbers = map(int, cleaned)

# Step 3: Sum the numbers

total = reduce(lambda x, y: x + y, numbers)

print(total)

# Output: 60

**3.2 Comprehensions for Clean Code**

In data processing and analysis, it's common to loop over data to **filter**, **transform**, or **extract** values

Python offers a powerful feature called **comprehensions**, which allow you to write **concise** (**fewer lines** and **less repetition**) and **expressive (clear and easy to understand)** code, replacing traditional for loops.

There are three main types:

* **List comprehensions**
* **Dictionary comprehensions**
* **Set comprehensions**

**3.2.1 List and Dictionary Comprehensions**

**List Comprehensions :**

A list comprehension allows you to create a new list by writing an expression inside square brackets with a for loop, and optionally an if condition.

* **Quickly creates a new list** by looping through an existing iterable (like a list or range).
* **Allows filtering and transforming** items in a single line of code.

Syntax :

[expression for item in iterable if condition]

expression: what you want to do with each item (e.g. x\*\*2)

iterable: the source list or data you're looping through

condition (optional): a filter that keeps only items that match

1. Create a list of squares

squares = [x\*\*2 for x in range(5)]

# Output: [0, 1, 4, 9, 16]

2. Filter even numbers

numbers = [1, 2, 3, 4, 5, 6]

evens = [x for x in numbers if x % 2 == 0]

# Output: [2, 4, 6]

3. Clean a list of strings

names = [' Alice ', 'BOB', ' Eve']

cleaned = [name.strip().lower() for name in names]

# Output: ['alice', 'bob', 'eve']

**Dictionary Comprehensions**

Dictionary comprehensions work the same way as list comprehensions, but they create key-value pairs using curly braces {}.

* **Quickly builds a dictionary** by defining how to create keys and values from an iterable.
* **Supports filtering and transformation** of key-value pairs.

Syntax

{key: value for item in iterable if condition}

key : What you want to use as the **key** in the new dictionary

value : What you want to use as the **value** in the dictionary

iterable : The data you're looping through — usually a list, tuple, or dictionary

condition : A filter to include only certain items

Examples

Create a dictionary of squares

squares = {x: x\*\*2 for x in range(5)}

# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Filter and modify a dictionary

original = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

filtered = {k: v\*10 for k, v in original.items() if v % 2 == 0}

# Output: {'b': 20, 'd': 40}

**3.2.2 Writing Compact Loops for Data Filtering and Transformation**

Comprehensions allow you to write compact (small) loops that handle both filtering and transformation in a single line — making your code cleaner and easier to maintain