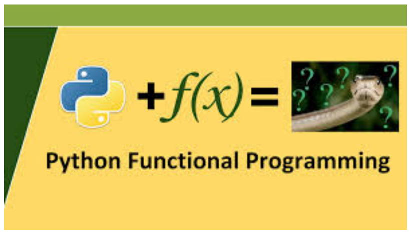
Functional Programming

Agenda

- Introduction to Functional Programming
    - What is Functional Programming?
    - Key Characteristics of Fuctional Programming
    - Benifits of Functional Programming
    - Comparison with Imperative and Object-Oriented Programming
- Function
- Core concept
- Lambda Functions
    - Diff b/w Lambda function vs def Keyword
- Decorators
- MAPS
- FILTERS
- ZIP
- REDUCE

Introduction to Functional Programming



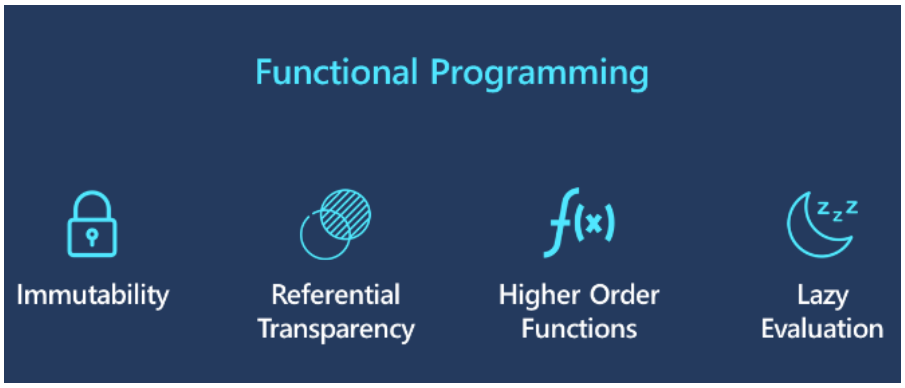
- Functional programming (FP) is a way of writing programs where you use functions to get things done, like solving math problems.
- It's different from the usual way of programming, which focuses on telling the computer how to do things step by step.

What is Functional Programming?

- Functional Programming (FP) is a style of programming where you use functions, just like in math, to perform tasks.
- It avoids changing data and keeps things simple by not allowing data to be modified once it is created.

- Functional programming (FP) is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.
- Unlike imperative programming, where the focus is on how to perform tasks (using loops, conditionals, and state changes), functional programming emphasizes what to do, using expressions that result in values.

Key Characteristics of Functional Programming



Pure Functions:

- A function is pure if its output is determined only by its input values, without observable side effects.
- Pure functions do not modify any external state or depend on it, ensuring that repeated calls with the same inputs produce the same outputs.
- Example: A function that calculates the area of a circle based only on the radius.

Immutability:

- In functional programming, data is immutable, meaning once a data structure is created, it cannot be altered.
- This leads to safer and more predictable code, as functions cannot change the data they operate on.
- Example: Once you create a shopping list, you cannot change it. Instead, you create a new list with the additional items.

First-Class and Higher-Order Functions:

- Functions are treated as first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.
- Higher-order functions are functions that take other functions as parameters or return functions as results.
- Example: You can give someone a function as a birthday gift, use a function as an ingredient in a recipe, or return a function as change from a transaction.

Function Composition:

- Function composition is the process of combining simple functions to build more complex ones.
- It promotes reusability and modularity in code by breaking down problems into smaller, composable pieces.
- Example: Building a complex machine by assembling smaller, simple parts.

Recursion:

- Recursion is the preferred method for iteration in functional programming, as loops are generally avoided.
- A function calls itself with a base condition to stop the recursive calls, replacing the need for iterative constructs like loops.
- Example: Finding your way out of a maze by taking a step, then solving the smaller maze in front of you, and so on.

Declarative vs. Imperative:

- Functional programming is declarative, meaning it focuses on what to compute rather than how to compute it.
- This contrasts with imperative programming, where the emphasis is on defining a sequence of instructions to change program state.
- Example: Telling someone to "cook dinner" (declarative) versus giving them a step-by-step recipe (imperative).

Benefits of Functional Programming

Functional programming (FP) offers a variety of advantages, particularly in terms of code clarity, maintainability, and reliability. Here are some key benefits:

1. Modularity and Reusability

- **Composable Functions**:
  - Functions are treated as first-class citizens, allowing them to be easily combined or reused in different parts of a program.
  - This leads to highly modular code where small, well-defined functions can be composed to create more complex functionality.
- **Separation of Concerns**:
  - Functional programming encourages breaking down a problem into smaller, independent functions, each responsible for a specific task.
  - This separation enhances code maintainability and makes it easier to understand and modify.

2. Immutability

- **Predictable State Management**:
  - In FP, data structures are immutable, meaning they cannot be changed after they are created.
  - This eliminates many of the bugs associated with mutable state, such as race conditions and unexpected side effects.
- **Thread-Safety**:
  - Immutability makes functional programs inherently safer for concurrent and parallel execution, as there's no risk of data being modified by multiple threads simultaneously.

3. Pure Functions

- **No Side Effects**:
  - Pure functions, which do not rely on or modify external state, make reasoning about code much easier.
  - Since the output of a pure function depends only on its input, it behaves consistently and predictably.
- **Ease of Testing**:
  - Testing pure functions is straightforward because they are isolated from external factors.
  - You simply provide inputs and verify that the outputs are as expected, without worrying about the function's interactions with external systems or state.

4. Declarative Code

- **Focus on *What* Not *How***:
  - Functional programming emphasizes describing *what* the program should accomplish, rather than specifying step-by-step instructions on *how* to achieve it.
  - This often leads to more concise and readable code, as the intent is clear and direct.
- **Easier Maintenance**:
  - Declarative code tends to be more maintainable, as it abstracts away the complexity of control flow and state management, reducing the cognitive load required to understand the program.

5. Parallelism and Concurrency

- **Simplicity in Concurrency**:
  - Functional programs, with their reliance on immutability and pure functions, are naturally well-suited for concurrent and parallel execution.

 - There's less need for complex locking mechanisms or synchronization, making it easier to write correct, concurrent code.
- **Scalability**:
  - Due to the ease of parallelizing functional code, it can scale more efficiently on multi-core processors, leading to better performance for computationally intensive tasks.

6. Reduced Bugs and Improved Reliability

- **Fewer Side Effects**:
  - By avoiding side effects and relying on pure functions, functional programming reduces the likelihood of bugs related to unexpected state changes or interactions between different parts of a program.
- **Referential Transparency**:
  - Since expressions in functional programming can be replaced with their corresponding values without changing the program's behavior, the code becomes more predictable and reliable.

7. Enhanced Code Readability and Maintainability

- **Clearer Code**:
  - The focus on pure functions, immutability, and declarative constructs often results in cleaner and more readable code, which is easier to understand and maintain.
- **Easier Refactoring**:
  - Functional code can often be refactored with minimal risk of introducing bugs, as the functions are independent and side-effect-free.

8. Better Abstractions

- **Higher-Order Functions**:
  - Functional programming allows for higher-order functions, which can abstract common patterns of computation, reducing code duplication and improving code structure.
- **Lazy Evaluation**:
  - Functional languages often support lazy evaluation, which means expressions are only evaluated when their results are needed.
  - This can lead to performance optimizations and the ability to work with infinite data structures.

9. Better Testability

- **Automated Testing**:
  - The deterministic nature of pure functions makes them ideal candidates for automated testing, including property-based testing, where the behavior of functions is tested against a wide range of inputs.
- **Simplified Debugging**:
  - Since functional code is often simpler and free of side effects, debugging becomes more straightforward, as there are fewer variables and states to track down.

10. Encourages a Functional Mindset

- **Problem-Solving Approach**:
  - Functional programming encourages thinking in terms of functions and transformations, which can lead to more elegant and efficient solutions to complex problems.
- **Functional Patterns**:

  - Adopting a functional mindset can help programmers recognize and apply functional patterns in other paradigms, leading to more robust and flexible code even outside of pure functional languages.

Comparison of Functional Programming with Imperative and Object-Oriented Programming

Functional programming (FP), imperative programming (IP), and object-oriented programming (OOP) are three distinct programming paradigms. Each has its own approach to solving problems, managing state, and structuring code.

1. Core Concepts

- **Functional Programming (FP)**:
  - Focuses on computation through mathematical functions.
  - Emphasizes *what* to do rather than *how* to do it.
  - Relies on pure functions, immutability, and function composition.
- **Imperative Programming (IP)**:
  - Focuses on *how* to perform tasks using a sequence of instructions that change program state.
  - Emphasizes step-by-step procedures and control flow, such as loops and conditionals.
  - Typical languages: C, C++, Python (in imperative style), JavaScript (in imperative style).
- **Object-Oriented Programming (OOP)**:
  - Organizes code around objects, which are instances of classes.
  - Emphasizes data encapsulation, inheritance, and polymorphism.
  - State and behavior are combined in objects, with methods manipulating an object's state.
  - Typical languages: Java, C++, Python (in OOP style), C#.

2. State Management

- **FP**:
  - Data is immutable, meaning once created, it cannot be changed.
  - State is passed through functions rather than being modified in place.
  - Side effects are minimized or eliminated, leading to more predictable code.
- **IP**:
  - State is mutable and often managed through variables that can be updated throughout the program.
  - Global and local variables are commonly used to maintain state.
  - Side effects are common, as functions and procedures modify global or external state.
- **OOP**:
  - State is encapsulated within objects.
  - Objects maintain their own state, and methods (functions) are used to modify this state.
  - Encapsulation helps manage complexity by hiding an object’s internal state from the outside world.

3. Code Structure

- **FP**:
  - Programs are structured as a collection of functions that transform data.
  - Functions are often small, focused, and can be composed together.
  - Code tends to be more declarative, describing *what* the program should do.
- **IP**:
  - Programs are structured around sequences of commands or statements.
  - Procedures or functions encapsulate blocks of code that perform specific tasks.
  - Code is generally more procedural and imperative, specifying *how* tasks should be accomplished.
- **OOP**:

- Programs are structured around classes and objects.
- Code is organized into methods within classes, with each object representing an instance of a class.
- Inheritance and polymorphism allow for code reuse and extension.

4. Abstraction Mechanisms

- **FP**:
  - Abstractions are achieved through higher-order functions, function composition, and pure functions.
  - Common functional constructs include map, filter, reduce, and recursion.
  - Monads, functors, and other abstract types are used to manage side effects and encapsulate computations.
- **IP**:
  - Abstraction is typically achieved through functions and procedures.
  - Control flow constructs like loops, conditionals, and error handling mechanisms are used.
  - Less emphasis on abstract data types compared to FP and OOP.
- **OOP**:
  - Abstractions are achieved through classes, inheritance, and polymorphism.
  - Interfaces and abstract classes define common behavior across different objects.
  - Design patterns (like Singleton, Factory, Observer) are commonly used to manage complexity.

5. Concurrency and Parallelism

- **FP**:
  - FP's emphasis on immutability and pure functions makes it well-suited for concurrent and parallel programming.
  - Without side effects, functions can run in parallel without the risk of data races or other concurrency issues.
- **IP**:
  - Concurrency is often managed through explicit thread management, locks, and synchronization mechanisms.
  - Mutable state can lead to challenges like race conditions, requiring careful handling.
- **OOP**:
  - Concurrency is typically managed through threading models, with synchronization mechanisms to protect shared state.
  - Mutable objects can lead to similar concurrency challenges as in imperative programming.

6. Testing and Debugging

- **FP**:
  - Pure functions and immutability make FP code easier to test and debug.
  - Since functions do not have side effects, they can be tested in isolation without concerns about external dependencies.
- **IP**:
  - Testing and debugging can be more challenging due to mutable state and side effects.
  - Functions that modify global or external state can lead to hidden dependencies, making it harder to isolate bugs.
- **OOP**:
  - Testing often requires setting up objects and their states.
  - Unit testing frameworks can help, but managing object state and interactions can introduce complexity.
  - Mocking and stubbing are common practices to isolate the behavior of objects in tests.

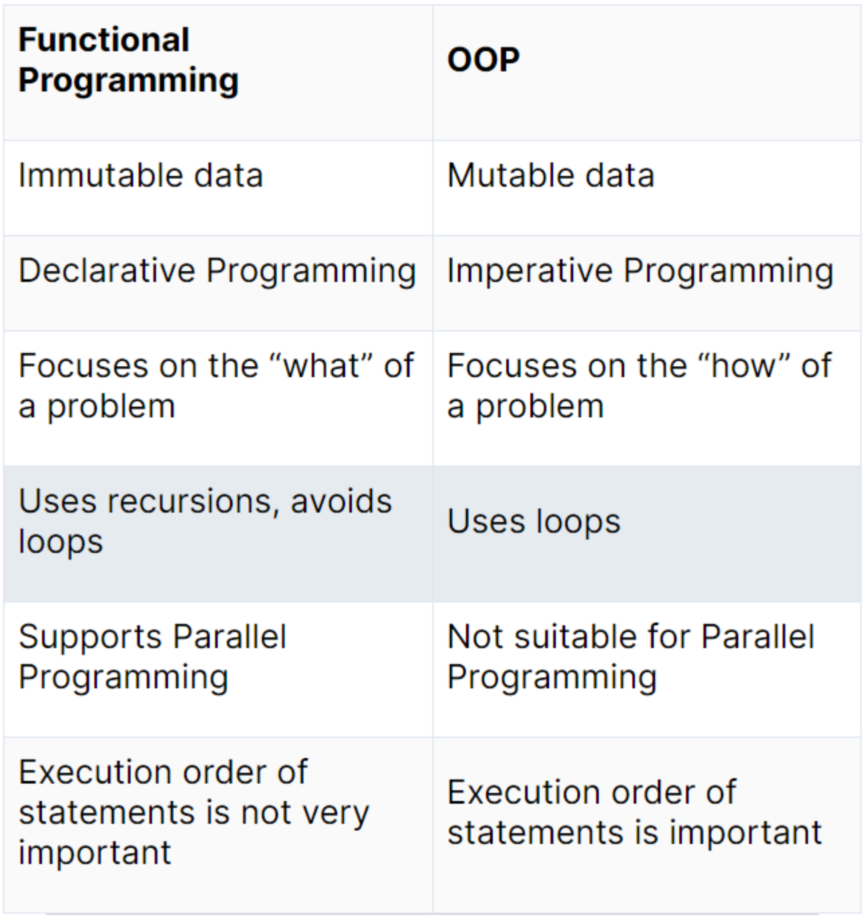
7. Learning Curve and Use Cases

- **FP**:
  - Can have a steep learning curve, especially for those unfamiliar with concepts like higher-order functions, recursion, and immutability.

    - Well-suited for data transformation tasks, parallel processing, and scenarios where predictable, bug-free code is critical.
- **IP**:
    - Generally considered easier to learn, especially for beginners, as it aligns closely with how computers operate at a low level.
    - Commonly used in system programming, scripting, and scenarios where fine-grained control over the program flow is needed.
- **OOP**:
    - The learning curve can vary; understanding objects, classes, inheritance, and design patterns takes time.
    - Ideal for large, complex applications where managing state and behavior through objects helps structure the code. Common in enterprise applications, GUI development, and game development.

8. Real-World Examples

- **FP**:
    - Functional programming is often used in data processing frameworks (e.g., Apache Spark), functional languages (e.g., Haskell, Scala), and concurrent systems (e.g., Elixir/Erlang).
- **IP**:
    - Imperative programming is used in systems programming (e.g., C for operating systems), scripting (e.g., Python for automation), and low-level embedded systems.
- **OOP**:
    - Object-oriented programming is prevalent in large-scale software development, such as enterprise software (e.g., Java, C#), GUI applications (e.g., JavaFX, PyQt), and mobile development (e.g., Swift for iOS, Kotlin for Android).



Functional Programming	OOP
Immutable data	Mutable data
Declarative Programming	Imperative Programming
Focuses on the “what” of a problem	Focuses on the “how” of a problem
Uses recursions, avoids loops	Uses loops
Supports Parallel Programming	Not suitable for Parallel Programming
Execution order of statements is not very important	Execution order of statements is important

Function

- Functions in Python are blocks of reusable code that perform a specific task.

- They help in organizing code, making it more modular, readable, and maintainable.

1. Defining a Function

- To define a function in Python, you use the `def` keyword followed by the function name, parentheses (which may contain parameters), and a colon.
- The function body is indented and contains the code that the function will execute.

```
def function_name(parameters):
    # Function body
    return value
```

- **`function_name`** : The name of the function.
- **`parameters`** : Optional. Variables passed to the function for processing.
- **`return`** : Optional. Specifies the value that the function returns.

2. Example of a Simple Function

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

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

```
Hello, Alice!
```

- **`greet`** is a function that takes a single argument `name` and returns a greeting string.

3. Calling a Function

To call a function, use its name followed by parentheses, and pass the necessary arguments if the function requires them.

```
result = greet("Bob")
print(result)  # Output: Hello, Bob!
```

```
Hello, Bob!
```

4. Parameters and Arguments

- **Positional Arguments**: The simplest form of passing arguments, where the order matters.

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

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

```
8
```

- **Keyword Arguments**: Arguments can be passed by explicitly naming them.

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

```
8
8
```

- **Default Arguments**: You can assign default values to parameters, which are used if no argument is provided.

```
  def greet(name="Guest"):
      return f"Hello, {name}!"
```

```
print(greet())         # Output: Hello, Guest!
print(greet("Alice"))  # Output: Hello, Alice!
```

```
Hello, Guest!
Hello, Alice!
```

- **Arbitrary Arguments (`*args`)**: Used when you don't know how many arguments will be passed.

```
def add(*args):
      return sum(args)

print(add(1, 2, 3))  # Output: 6
```

```
6
```

- **Arbitrary Keyword Arguments (`kwargs`)**: Used when you want to pass a variable number of keyword arguments.

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=30)  # Output: name: Alice, age: 30
```

```
name: Alice
age: 30
```

5. Returning Values

- A function can return a value using the `return` statement.
- If no return statement is used, the function returns `None` .

```
def square(x):
    return x * x

result = square(4)
print(result)  # Output: 16
```

```
16
```

7. Scope of Variables

- Variables defined inside a function are local to that function and cannot be accessed outside of it.

```
def example():
    x = 10  # Local variable
    return x

print(example())  # Output: 10
# print(x)  # This would raise an error since x is not defined outside the function
```

```
10
```

8. Nested Functions

- Functions can be defined inside other functions. These are called nested functions.

```
def outer_function():
    def inner_function():
        return "Hello from the inner function!"
    return inner_function()

print(outer_function())  # Output: Hello from the inner function!
```

```
Hello from the inner function!
```

9. Recursion

- A function can call itself. This is known as recursion. It's useful for problems that can be divided into similar subproblems.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

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

```
120
```

10. Docstrings

- You can add documentation to a function using a docstring, which is a string literal placed as the first statement in a function.

```
def greet(name):
    """This function returns a greeting message."""
    return f"Hello, {name}!"

print(greet.__doc__)  # Output: This function returns a greeting message.
```

```
This function returns a greeting message.
```

11. Function Annotations

- Python allows you to add metadata to function parameters and return values using annotations.

```
def add(a: int, b: int) -> int:
    return a + b

print(add(3, 5))  # Output: 8
print(add.__annotations__)  # Output: {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

```
8
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Core concept

Pure Functions

A pure function is a fundamental concept in functional programming that adheres to specific criteria, making it predictable, testable, and reliable.

1. Definition of Pure Functions

- **Deterministic Output**:
  - A pure function always produces the same output for the same set of inputs, no matter when or where it is called. This is often referred to as determinism.

- **No Side Effects**:
  - A pure function does not cause any observable side effects outside of returning a value.
  - This means it doesn't modify any external state, such as variables, files, or data structures, and it doesn't perform operations like logging, printing, or writing to a database.

2. Characteristics of Pure Functions

- **Referential Transparency**:
  - Pure functions are referentially transparent, meaning any expression in a program that calls a pure function can be replaced with the function's result without changing the program's behavior.
  - This property makes it easier to reason about and refactor code.
- **No Dependency on External State**:
  - Pure functions do not rely on any external state or mutable data.
  - They only use the data passed to them as arguments, ensuring their behavior is consistent and predictable.

Examples of Pure Functions

- **Pure Function Example (Python)**:

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

  In this example, the function `add` is pure because it always returns the same result for the same inputs and does not modify any external state.

- **Impure Function Example (Python)**:

```
result = 0

def add_to_result(a):
    global result
    result += a
    return result
```

  Here, the function `add_to_result` is impure because it modifies the global variable `result`, causing a side effect. The function's output depends on the external state of `result`, making it impure.

Benefits of Pure Functions

- **Ease of Testing**: Since pure functions do not rely on or modify external state, they can be easily tested in isolation. You can simply provide inputs and verify that the output matches the expected result without worrying about side effects or dependencies.
- **Predictability and Reliability**: Pure functions are predictable, meaning they always behave the same way for the same inputs. This predictability makes them more reliable and easier to debug.
- **Simplified Concurrency**: Pure functions are naturally thread-safe, as they do not modify shared state or depend on mutable data. This makes them well-suited for concurrent or parallel programming, where managing shared state can be challenging.
- **Referential Transparency**: The property of referential transparency allows for easier reasoning about code. You can replace a function call with its result, simplifying the understanding and refactoring of code.

Limitations of Pure Functions

- **Interacting with the Real World**:
  - Pure functions are limited when it comes to interacting with the real world, such as reading from or writing to a file, generating random numbers, or accessing a database.
  - These operations inherently involve side effects, so they cannot be purely functional.

- **Performance Considerations**:
  - In some cases, the use of pure functions may lead to performance trade-offs, especially when immutability and lack of side effects require additional computation or memory usage.
  - However, these trade-offs are often outweighed by the benefits of maintainability and correctness.

Pure Functions in Functional Programming Languages

- **Haskell**: Haskell enforces purity at the language level. All functions in Haskell are pure by default, and side effects are managed using monads, such as the `IO` monad.
- **Scala**: Scala encourages the use of pure functions, especially when using functional programming libraries and constructs, although it also supports imperative and object-oriented styles.
- **Elixir**: Elixir promotes the use of pure functions and immutability, making it a good choice for building concurrent and distributed systems.

```
def multiply(a, b):
    return a * b



def concatenate_strings(str1, str2):
    return str1 + str2



def add_to_list(lst, element):
    return lst + [element]



def filter_even_numbers(numbers):
    return [num for num in numbers if num % 2 == 0]



def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)



def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

First-Class and Higher-Order Functions

First-Class Functions and **Higher-Order Functions** are key concepts in functional programming that allow functions to be treated as values and manipulated just like other data types.

First-Class Functions

First-Class Functions mean that functions in a programming language are treated as first-class citizens. This means they can be:

- **Assigned to variables**: You can assign a function to a variable.
- **Passed as arguments**: You can pass functions as arguments to other functions.
- **Returned from other functions**: A function can return another function.
- **Stored in data structures**: Functions can be stored in lists, dictionaries, etc.

Examples of First-Class Functions in Python

1. **Assigning a Function to a Variable**:

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

say_hello = greet  # Assigning the function to a variable
print(say_hello("Alice"))  # Output: Hello, Alice!
```

2. **Passing a Function as an Argument**:

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

def process_function(func, name):
    return func(name)

print(process_function(greet, "Bob"))  # Output: Hello, Bob!
```

3. **Returning a Function from Another Function**:

```
def outer_function():
    def inner_function():
        return "Hello from the inner function!"
    return inner_function

func = outer_function()
print(func())  # Output: Hello from the inner function!
```

4. **Storing Functions in a Data Structure**:

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

def subtract(x, y):
    return x - y

operations = {
    "add": add,
    "subtract": subtract
}

print(operations["add"](5, 3))  # Output: 8
print(operations["subtract"](5, 3))  # Output: 2
```

Higher-Order Functions

A **Higher-Order Function** is a function that either:

- Takes one or more functions as arguments, or
- Returns another function as its result.

Higher-order functions are powerful tools in functional programming, allowing for greater abstraction and reusability of code.

Examples of Higher-Order Functions in Python

1. **Map Function**:

   The `map()` function is a built-in higher-order function that applies a function to each item of an iterable (like a list) and returns a map object (which can be converted to a list).

```
def square(x):
    return x * x

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

2. **Filter Function**:

   The `filter()` function is a higher-order function that filters elements from an iterable based on a function that returns a Boolean value.

```
def is_even(x):
    return x % 2 == 0

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

3. **Reduce Function**:

   The `reduce()` function (from `functools` module) is a higher-order function that applies a binary function cumulatively to the items of an iterable, reducing the iterable to a single value.

```
from functools import reduce

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

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(add, numbers)
print(sum_of_numbers)  # Output: 15
```

4. **Returning Functions from Functions**:

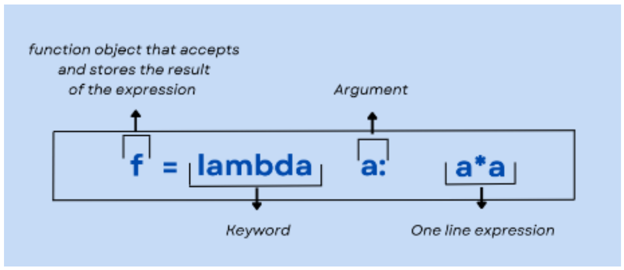
```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
print(add_five(10))  # Output: 15
```

5. **Lambda Functions as Higher-Order Functions**:

   Lambda functions are often used in Python as arguments to higher-order functions.

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

Lambda Functions


function object that accepts and stores the result of the expression
Argument
f = lambda a: a*a
Keyword
One line expression

- Lambda functions, often called anonymous functions, are a concise way to create small, inline functions without the need for a formal function definition.
- They are an essential tool in functional programming and find various use cases in Data Science and Machine Learning (DSML) for tasks such as data preprocessing.

- Lambda functions are small, unnamed functions defined using the lambda keyword.
- They are typically used for simple operations and are often employed when a function is required for a short period and doesn't need a formal name.
- The general syntax of a lambda function is as follows:

```
lambda arguments: expression
```

Characteristics of Lamda Fucntions

- They can take any number of arguments but can contain only a single expression.
- They are concise and primarily used for simple operations.
- Lambda functions are often used in combination with higher-order functions like map, filter, and reduce.

Use Cases in Data Preprocessing

Lambda functions play a significant role in data preprocessing in DSML.

Here are some common use cases:

1. **Data Transformation**:
   - Lambda functions are used with functions like map and apply to transform data.
   - For instance, you can use a lambda function to convert data types, apply mathematical operations, or perform text manipulation.
2. **Filtering Data**:
   - In data cleaning and filtering tasks, lambda functions are employed with filter to select specific data points that meet certain conditions.
   - For example, filtering out outliers from a dataset.
3. **Feature Engineering**:
   - Lambda functions assist in creating new features or variables by combining or transforming existing ones.
   - They can be used to calculate ratios, perform mathematical operations, or apply custom logic to create new features.
4. **Sorting and Aggregating**:
   - Lambda functions can be used as keys for sorting data, especially when you want to sort based on a specific attribute or criterion.
   - They are also useful when aggregating data using functions like groupby in Pandas.

Examples of Lambda Functions

Python Lambda Function with List Comprehension

In this example, we use a lambda function to create a list:

```
simple_list = [lambda arg=x: arg for x in range(1, 11)]

for item in simple_list:
    print(item())
```

```
1
2
3
4
5
6
```

```
7
8
9
10
```

Python Lambda Function with if-else

This example uses a lambda function with an if-else statement to find the minimum of two numbers:

```
min = lambda a, b: a if a < b else b

print(min(4, 12))
```

```
4
```

Using lambda() Function with filter()

This example filters out all the even numbers from a list:

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filtered_list = list(filter(lambda x: (x % 2 != 1), l))

print(filtered_list)
```

```
[2, 4, 6, 8, 10]
```

Using lambda() Function with map()

In this example, we use map() to update all the elements of an array:

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

final_list = list(map(lambda x: x + 10, l))

print(final_list)
```

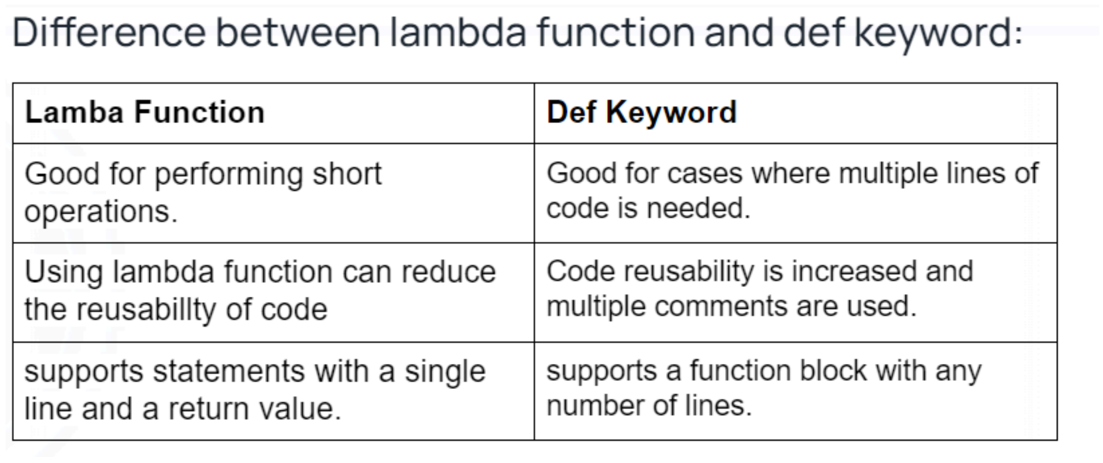
```
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
# Application 1: Lambda Function with List Comprehension

print("Example 1: Lambda Function with List Comprehension")

# Creating a list of lambda functions using list comprehension
simple_list = [lambda arg=x: arg for x in range(1, 11)]

# Iterating over the list and calling each lambda function
for item in simple_list:
    print(item())  # Output: 1, 2, 3, ..., 10

print("\n")

# Application 2: Lambda Function with if-else

print("Example 2: Lambda Function with if-else")

# Lambda function to find the minimum of two numbers
min_func = lambda a, b: a if a < b else b

# Using the lambda function
result = min_func(4, 12)
print(f"Minimum of 4 and 12: {result}")  # Output: 4

print("\n")

# Application 3: Using lambda() function with filter()

print("Example 3: Using lambda() function with filter()")

# List of numbers
```

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filtering out all even numbers using filter() and lambda
filtered_list = list(filter(lambda x: x % 2 == 0, l))

print(f"Even numbers from the list: {filtered_list}")  # Output: [2, 4, 6, 8, 10]

print("\n")

# Application 4: Using lambda() function with map()

print("Example 4: Using lambda() function with map()")

# Updating all elements of the list by adding 10 using map() and lambda
final_list = list(map(lambda x: x + 10, l))

print(f"List after adding 10 to each element: {final_list}")  # Output: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

print("\n")
```

```
Example 1: Lambda Function with List Comprehension
1
2
3
4
5
6
7
8
9
10


Example 2: Lambda Function with if-else
Minimum of 4 and 12: 4


Example 3: Using lambda() function with filter()
Even numbers from the list: [2, 4, 6, 8, 10]


Example 4: Using lambda() function with map()
List after adding 10 to each element: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

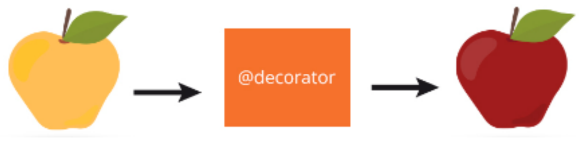
Diff b/w Lambda function vs def Keyword



Difference between lambda function and def keyword:

Lamba Function	Def Keyword
Good for performing short operations.	Good for cases where multiple lines of code is needed.
Using lambda function can reduce the reusabillty of code	Code reusability is increased and multiple comments are used.
supports statements with a single line and a return value.	supports a function block with any number of lines.

Decorators



- Decorators are powerful tools in Python, especially valuable in Data Science and Machine Learning (DSML) for improving code readability and reusability.
- They enable you to modify function behavior without changing the core code, enhancing modularity.

What Are Decorators?

Decorators are functions that take other functions as input and modify their behavior.

In DSML, decorators are essential because they:

1. **Promote Modularity:** Separate concerns, making it easy to apply specific functionality to different functions or methods.
2. **Enhance Readability:** Streamline code by abstracting repetitive or non-essential logic, resulting in cleaner, focused functions.

Creating and Applying Decorators

To use decorators effectively:

1. **Define the Decorator Function:**
   - Create a regular Python function that takes another function as an argument.
   - Customize the code inside the decorator function to enhance the behavior of the input function.
2. **Apply the Decorator:**
   - Prefix a function definition with "@" followed by the decorator name.
   - This tells Python to apply the decorator to the function.
3. **Use Decorated Functions:**
   - Call the decorated function like any regular function.
   - The decorator's code runs alongside the original function's code.

Basic Syntax of a Decorator

A decorator is typically a function that takes another function as an argument, adds some kind of functionality, and then returns a new function. Here's a simple example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

How Decorators Work

- **Function as Argument:** The decorator function (my_decorator) takes another function (say_hello) as its argument.
- **Wrapper Function:** Inside the decorator, a new function (wrapper) is defined.This function adds functionality before and after calling the original function.
- **Return Value:** The decorator returns the wrapper function, effectively replacing the original function with the new one.
- **Using @decorator Syntax:** The @my_decorator syntax is a shorthand for applying the decorator to the function.

Decorators with Arguments

Decorators can also take arguments if you want to customize their behavior:

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Vaibhav")
```

```
Hello, Vaibhav!
Hello, Vaibhav!
Hello, Vaibhav!
```

Built-in Decorators

Python has several built-in decorators like @staticmethod, @classmethod, and @property, which are commonly used in class definitions.

1. @staticmethod

The @staticmethod decorator is used to define a method that doesn't require access to the instance (self) or the class (cls). It behaves like a regular function, but it belongs to the class's namespace and can be called on an instance or directly on the class.

```
class MyClass:
    @staticmethod
    def my_static_method():
        print("This is a static method.")

# Calling the static method
MyClass.my_static_method()

# You can also call it using an instance
obj = MyClass()
obj.my_static_method()
```

```
This is a static method.
This is a static method.
```

2. @classmethod

The @classmethod decorator is used to define a method that receives the class itself as the first argument, typically named cls. This is useful for methods that need to modify class-level data or need access to class-level methods.

```
class MyClass:
    count = 0

    @classmethod
    def increment_count(cls):
        cls.count += 1

# Calling the class method
MyClass.increment_count()
print(MyClass.count)

# You can also call it using an instance
obj = MyClass()
obj.increment_count()
print(MyClass.count)
```

```
1
2
```

3. @property

The @property decorator is used to define a method that acts like an attribute. This allows you to define "getter" methods in a way that they can be accessed as attributes, making the code cleaner and more intuitive.

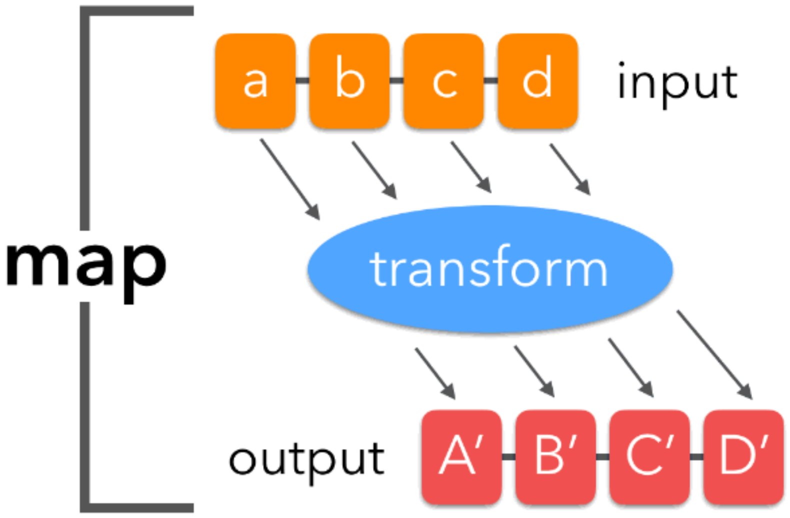
```
class MyClass:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        if new_value >= 0:
            self._value = new_value
        else:
            raise ValueError("Value must be non-negative")

# Using the property
obj = MyClass(10)
print(obj.value)  # Calls the getter method

obj.value = 20    # Calls the setter method
print(obj.value)

# Trying to set a negative value will raise an error
# obj.value = -5   # Uncommenting this line will raise ValueError
```

```
10
20
```

Use Cases

- **@staticmethod:** Use when you need a method that does not modify class or instance state.
- **@classmethod:** Use when you need a method that modifies class state or needs to work with the class rather than an instance.
- **@property:** Use when you want to control access to instance attributes, providing a clean interface for "getter" and "setter" methods.

MAPS



- The map function applies a given function to each item in an iterable (e.g., a list) and returns a new iterable with the results.
- It's an efficient way to transform data, applying the same operation to multiple elements.

Basic Syntax

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

- **function**: The function to apply to each element of the iterable.
- **iterable**: One or more iterable(s) (like lists, tuples, etc.) to which the function will be applied.

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)

# Converting map object to list for easier viewing
print(list(squared_numbers))
```

```
[1, 4, 9, 16, 25]
```

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

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
result = map(add, numbers1, numbers2)

print(list(result))
```

```
[5, 7, 9]
```

Map with lamda Function

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

```
[1, 4, 9, 16, 25]
```

Converting map Object to a List

The result of map() is an iterator, so if you want to see all the results at once, you need to convert it to a list:

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

```
celsius = [0, 10, 20, 30, 40]
fahrenheit = map(lambda x: (x * 9/5) + 32, celsius)
print(list(fahrenheit))
```

```
[32.0, 50.0, 68.0, 86.0, 104.0]
```

Lazy Evaluation with map()

map() returns a lazy iterator, which means the transformation is only applied when you iterate over the map object. This is useful for working with large datasets, as it saves memory by not generating all results at once.

```
def double(x):
    return x * 2

numbers = range(10**6)  # A large range
doubled_numbers = map(double, numbers)

# Only the first 10 elements are computed here
print(list(doubled_numbers)[:10])
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Using map() with functools.partial

The functools.partial function can be used to partially apply a function, making it easier to use with map() when you have a function that takes multiple arguments.

```
from functools import partial

def multiply(x, y):
    return x * y

# Create a new function that multiplies by 2
double = partial(multiply, 2)

numbers = [1, 2, 3, 4, 5]
result = map(double, numbers)
print(list(result))
```

```
[2, 4, 6, 8, 10]
```

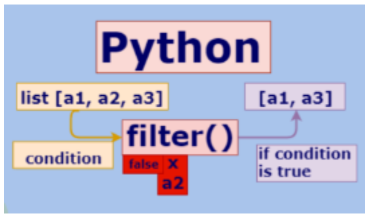
Chaining map() with Other Functional Constructs

You can chain map() with other functional constructs like filter() and reduce() to create more complex data processing pipelines.

```
from functools import reduce

numbers = range(1, 11)

# Double the numbers, filter out the even ones, and then sum them
result = reduce(lambda x, y: x + y, filter(lambda x: x % 2 == 0, map(lambda x: x * 2, numbers)))
print(result)
```

```
110
```

FILTERS



The filter function filters elements from an iterable based on a specified condition. It's useful for selecting specific data points that meet certain criteria.

Basic Syntax

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

- function: A function that tests each element of the iterable. It should return True to include the element in the result and False otherwise.
- iterable: The iterable (like a list, tuple, or string) whose elements you want to filter.

Filtering Even Numbers from a List

```
def is_even(num):
    return num % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(is_even, numbers)

# Converting the filter object to a list for easier viewing
print(list(even_numbers))
```

```
[2, 4, 6, 8, 10]
```

Using 'filter()' with Lambda Functions

You can use lambda functions with filter() to simplify the syntax:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))
```

```
[2, 4, 6, 8, 10]
```

```
words = ["apple", "banana", "cherry", "date", "fig", "grape"]
long_words = filter(lambda word: len(word) > 4, words)

print(list(long_words))
```

```
['apple', 'banana', 'cherry', 'grape']
```

Using filter() with Multiple Iterables

Although filter() is typically used with a single iterable, you can combine it with functions like zip() to filter based on multiple iterables.

```
names = ["Alice", "Bob", "Charlie", "Diana"]
ages = [25, 30, 17, 20]

# Filter out names of people who are 21 or older
adults = filter(lambda x: x[1] >= 21, zip(names, ages))
print(list(adults))
```

```
[('Alice', 25), ('Bob', 30)]
```

Filtering with Custom Objects

You can use filter() to process custom objects if the function provided is designed to handle the object's attributes.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

people = [
    Person("Alice", 25),
    Person("Bob", 17),
    Person("Charlie", 19),
    Person("Diana", 23)
]

# Filter out people who are 21 or older
adults = filter(lambda person: person.age >= 21, people)

# Extract names of adults
adult_names = map(lambda person: person.name, adults)
print(list(adult_names))
```

```
['Alice', 'Diana']
```

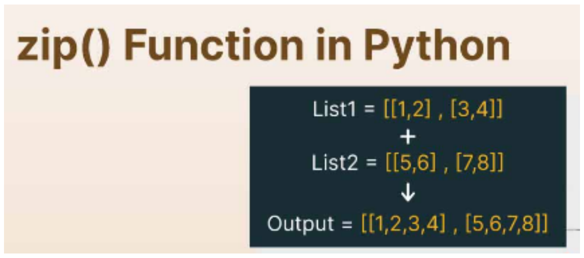
Chaining filter() with map() and reduce()

You can chain filter() with map() and reduce() to perform more complex data transformations and aggregations.

```
from functools import reduce

numbers = range(1, 21)

# Filter even numbers, square them, and then sum them up
result = reduce(lambda x, y: x + y, map(lambda x: x * x, filter(lambda x: x % 2 == 0, numbers)))
print(result)
```

```
1540
```

ZIP



The zip function in Python is a versatile tool for combining multiple iterables into a single iterable, creating pairs or tuples from corresponding elements of the input sequences. It takes two or more sequences as input and returns an iterator that generates tuples, where each tuple contains elements from the input sequences at the same index.

Basic Syntax

```
zip(*iterables)
```

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

```
### Handling Unequal length
list1 = [1, 2, 3, 4]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

```
###zip() with Multiple Iterables
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
list3 = [True, False, True]

zipped = zip(list1, list2, list3)
print(list(zipped))
```

```
[(1, 'a', True), (2, 'b', False), (3, 'c', True)]
```

Basic Operations with ZIP Files

1. Creating a ZIP File You can create a ZIP file by adding multiple files or an entire directory to it.

```
import zipfile

# List of files to be zipped
files_to_zip = ['file1.txt', 'file2.txt', 'file3.txt']

# Create a new ZIP file
with zipfile.ZipFile('my_archive.zip', 'w') as zipf:
    for file in files_to_zip:
        zipf.write(file)
```

Zipping an Entire Directory

```
import zipfile
import os

def zipdir(path, ziph):
    # Zip the entire directory
    for root, dirs, files in os.walk(path):
        for file in files:
            ziph.write(os.path.join(root, file),
                       os.path.relpath(os.path.join(root, file),
                       os.path.join(path, '..')))

# Create a ZIP file
with zipfile.ZipFile('my_archive.zip', 'w') as zipf:
    zipdir('my_directory', zipf)
```

Extracting ZIP Files

You can extract files from a ZIP archive using the extractall() or extract() method.

Example: Extracting All Files from a ZIP Archive

```
import zipfile

# Extract all files from a ZIP file
with zipfile.ZipFile('my_archive.zip', 'r') as zipf:
    zipf.extractall('extracted_files')
```

Listing Contents of a ZIP File

You can list the contents of a ZIP file without extracting them.

```
import zipfile

# List the contents of a ZIP file
with zipfile.ZipFile('my_archive.zip', 'r') as zipf:
    print(zipf.namelist())
```

```
[]
```

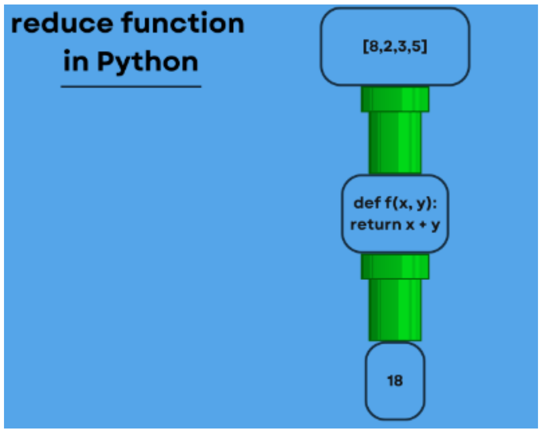
unzip

unzip is not a built-in function in Python, but it refers to the process of separating a zipped object back into individual iterables. This can be done using the zip() function with the unpacking operator (*).

```
zipped = [('a', 1), ('b', 2), ('c', 3)]

# Unzipping
letters, numbers = zip(*zipped)
print(letters)
print(numbers)
```

```
('a', 'b', 'c')
(1, 2, 3)
```

REDUCE



The reduce function (from the functools module) aggregates elements in an iterable using a specified function. It's often used for cumulative operations, like summing all elements in a list.

```
from functools import reduce

reduce(function, iterable[, initializer])
```

- function: A function that takes two arguments. This function is applied cumulatively to the items of the iterable.
- iterable: The iterable whose elements are to be reduced.
- initializer (optional): An initial value that can be passed to the function. If provided, it is placed before the items of the iterable in the calculation.

How reduce() Works

reduce() applies the function to the first two elements of the iterable. Then, it applies the function to the result of that and the next element. This process continues until the iterable is exhausted.

```
from functools import reduce

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

# Define a function for summation
def add(x, y):
    return x + y

result = reduce(add, numbers)
print(result)
```

```
15
```

Using a Lambda Function

You can use a lambda function to simplify the code:

```
from functools import reduce

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

# Use a lambda function for summation
result = reduce(lambda x, y: x + y, numbers)
print(result)
```

```
15
```

```
## Finding the Maximum Value
## You can use reduce() to find the maximum value in a list:

from functools import reduce

numbers = [3, 5, 2, 8, 1]

# Define a function to find the maximum
max_value = reduce(lambda x, y: x if x > y else y, numbers)
print(max_value)
```

```
8
```

```
## Concatenating Strings
## You can use reduce() to concatenate a list of strings into a single string:
from functools import reduce

words = ["Python", "is", "fun"]

# Define a function for concatenation
sentence = reduce(lambda x, y: x + " " + y, words)
```

```
print(sentence)
```

```
Python is fun
```

Using reduce() with an Initializer

You can provide an initializer to reduce(), which is useful when you want to start with a particular value.

```
from functools import reduce

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