**Python Task**

**Functional Programming**

Functional Programming (FP) is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Python supports multiple paradigms, including functional programming. Below is an overview highlighting several aspects of functional programming within Python:

**1. First-Class and High-Order Functions**

Python supports the concept of First-Class functions (functions can be passed around as arguments to other functions), and High-Order functions (functions that return another function).

def greet():

return "Hello!"

def caller(func):

return func()

print(caller(greet)) # Output: Hello!

**2. Pure Functions**

Pure functions have no side effects and return a value that depends only on their arguments.

def add(a, b):

return a + b

**3. Lambda Functions**

Lambda functions are anonymous functions defined using the `lambda` keyword. They are often used for short-duration, non-reusable operations.

multiply = lambda x, y: x \* y

print(multiply(3, 4)) # Output: 12

**4. Map, Filter, and Reduce**

These are higher-order functions that allow for functional-style data manipulation.

**Map**; Applies a function to all items in an input list (or another iterable).

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

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

**Filter**; Creates a list of elements for which a function returns true.

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

**Reduce**; Applies a rolling computation (function) to sequential pairs of values in a list.

from functools import reduce

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

**5. Immutability**

In FP, once a data structure is created, it cannot be changed. If you want to make a change, you create a new data structure. This is known as immutability and is seen in Python with tuples.

immutable\_tuple = (1, 2, 3)

**6. Recursion**

Functional programming favors recursive functions as the primary mechanism for performing repetitive tasks.

def factorial(n):

return 1 if n == 0 else n \* factorial(n-1)

**7. List Comprehensions**

Python's list comprehensions and generator expressions also have a functional flavor. They allow you to create a new list by applying an expression to each item in an existing collection.

squares = [x\*x for x in range(10)]

**Benefits and Drawbacks**

**Benefits**: Easier debugging (due to absence of side effects), easier testing, and enhanced readability and maintainability.

**Drawbacks**: Can be unfamiliar or slightly less intuitive for those accustomed to other paradigms, and sometimes recursion or higher-order functions can be computationally expensive or memory-intensive.

Despite not being a functional language, Python offers useful functional programming features, making it flexible and versatile in various use-cases and allows developers to utilize functional programming principles where they see fit.

**Lamba**

In Python, a `lambda` function is a small anonymous function. It is defined using the `lambda` keyword, followed by a list of arguments, a colon, and an expression:

lambda arguments: expression

**Characteristics:**

* **Anonymous**: `lambda` functions are anonymous, meaning they do not require a name to be used immediately and can be used in-line.
* **Simple**: They are typically used for short, simple operations that are easy to read and understand at a glance.
* **Single** **Expression**: Can only have a single expression, not a block of statements.

**Common Usage:**

* Functional Arguments\*\*: Often used as an argument to higher-order functions (functions that take in other functions as parameters) like `map()`, `filter()`, and `reduce()`.

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

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

* **Short-term Use**: Generally used for one-time, non-reusable operations where defining a full function would be syntactically over-verbose.
* **Callback or Key Functions**: Commonly used in sorting operations where you want to sort based on a derived key.

names = ['Alice', 'Bob', 'Charlie']

names.sort(key=lambda s: len(s))

Limitations:

* Simplicity: Designed for simple operations, trying to do too much in a `lambda` can make code less readable.
* Single Expression: Cannot include statements or multiple expressions.

In essence, `lambda` functions offer a concise way to perform small, simple operations where a full function definition would feel unnecessary or cumbersome.