**Functional Programming in Python -**

Key attributes of functional programming in python

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. Python is not a purely functional programming language, but it supports many functional programming features.

Below are the \*key attributes of functional programming in Python:  
  
---  
  
### 1. \*\*First-Class Functions\*\*  
 - Functions in Python are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.  
 - Example:  
 ```python  
 def square(x):  
 return x \* x  
  
 f = square # Assign function to a variable  
 print(f(5)) # Output: 25  
 ```  
  
---  
  
### 2. \*\*Higher-Order Functions\*\*  
 - Functions that take other functions as arguments or return functions as results are called higher-order functions.  
 - Common examples in Python: `map()`, `filter()`, and `reduce()`.  
 - Example:  
 ```python  
 def add\_one(x):  
 return x + 1  
  
 numbers = [1, 2, 3, 4]  
 result = map(add\_one, numbers) # Applies add\_one to each element  
 print(list(result)) # Output: [2, 3, 4, 5]  
 ```  
  
---  
  
### 3. \*\*Immutability\*\*  
 - Functional programming emphasizes immutability, where data structures are not modified after creation. While Python does not enforce immutability, you can use immutable types like tuples or `frozenset`.  
 - Example:  
 ```python  
 my\_tuple = (1, 2, 3)  
 # my\_tuple[0] = 10 # This would raise an error because tuples are immutable  
 ```  
  
---  
  
### 4. \*\*Pure Functions\*\*  
 - A pure function is a function where the output depends only on the input arguments and has no side effects (e.g., modifying global variables or I/O operations).  
 - Example:  
 ```python  
 def pure\_function(x, y):  
 return x + y # No side effects  
  
 print(pure\_function(2, 3)) # Output: 5  
 ```  
  
---  
  
### 5. \*\*Recursion\*\*  
 - Functional programming often uses recursion instead of loops for iteration. Python supports recursion, but it has a recursion depth limit.  
 - Example:  
 ```python  
 def factorial(n):  
 if n == 0:  
 return 1  
 return n \* factorial(n - 1)  
  
 print(factorial(5)) # Output: 120  
 ```  
  
---  
  
### 6. \*\*Anonymous Functions (Lambdas)\*\*  
 - Python supports anonymous functions using the `lambda` keyword. These are small, inline functions often used with higher-order functions.  
 - Example:  
 ```python  
 numbers = [1, 2, 3, 4]  
 result = map(lambda x: x \* 2, numbers)  
 print(list(result)) # Output: [2, 4, 6, 8]  
 ```  
  
---  
  
### 7. \*\*Function Composition\*\*  
 - Functional programming allows combining multiple functions to create new functions. While Python does not have built-in support for function composition, it can be achieved manually.  
 - Example:  
 ```python  
 def double(x):  
 return x \* 2  
  
 def increment(x):  
 return x + 1  
  
 def compose(f, g):  
 return lambda x: f(g(x))  
  
 composed\_function = compose(double, increment)  
 print(composed\_function(3)) # Output: 8 (double(increment(3)))  
 ```  
  
---  
  
### 8. \*\*Lazy Evaluation\*\*  
 - Functional programming often uses lazy evaluation to improve performance. In Python, this is achieved using generators or functions like `map()` and `filter()`, which do not compute results until needed.  
 - Example:  
 ```python  
 def generate\_numbers():  
 for i in range(5):  
 yield i # Lazy evaluation  
  
 gen = generate\_numbers()  
 print(next(gen)) # Output: 0  
 print(next(gen)) # Output: 1  
 ```  
  
---  
  
### 9. \*\*Declarative Style\*\*  
 - Functional programming emphasizes a declarative style, focusing on "what to do" rather than "how to do it." Python's list comprehensions and functional tools like `map()` and `filter()` support this style.  
 - Example:  
 ```python  
 numbers = [1, 2, 3, 4]  
 even\_numbers = filter(lambda x: x % 2 == 0, numbers)  
 print(list(even\_numbers)) # Output: [2, 4]  
 ```  
  
---  
  
### 10. \*\*Built-in Functional Tools\*\*  
 - Python provides several built-in tools and modules to support functional programming:  
 - \*\*`map()`\*\*: Applies a function to all items in an iterable.  
 - \*\*`filter()`\*\*: Filters items in an iterable based on a condition.  
 - \*\*`reduce()`\*\* (from `functools`): Reduces an iterable to a single value by applying a function cumulatively.  
 - \*\*`functools`\*\*: Provides utilities like `partial` and `lru\_cache`.  
 - \*\*`itertools`\*\*: Offers tools for working with iterators.  
 - Example:  
 ```python  
 from functools import reduce  
  
 numbers = [1, 2, 3, 4]  
 result = reduce(lambda x, y: x + y, numbers)  
 print(result) # Output: 10  
 ```  
  
---  
  
### Summary Table of Key Attributes

|  |  |
| --- | --- |
| **\*\*Attribute\*\*** | **\*\*Description\*\*** |
| First-Class Functions | Functions can be treated as variables. |
| Higher-Order Functions | Functions that take/return other functions. |
| Immutability | Emphasis on immutable data structures. |
| Pure Functions | Functions with no side effects. |
| Recursion | Using recursion instead of loops. |
| Anonymous Functions | Inline functions using `lambda`. |
| Function Composition | Combining functions to create new ones. |
| Lazy Evaluation | Delaying computation until needed (e.g., generators). |
| Declarative Style | Focus on "what to do" rather than "how to do it." |
| Built-in Functional Tools | Tools like `map()`, `filter()`, `reduce()`, and modules like `functools`. |

---  
  
By combining these attributes, Python allows developers to write clean, concise, and functional-style code, even though it is not a purely functional language.

A paradigm that emphasizes the use of pure mathematical functions, immutability, and declarative programming. Below is a summarized breakdown of the key concepts and examples provided:  
  
---  
  
### \*\*Key Concepts of Functional Programming\*\*  
1. \*\*Pure Functions\*\*:  
 - Always produce the same output for the same input.  
 - Do not modify input arguments or global variables (no side effects).  
 - Example:

def pure\_func(List):  
 return [i\*\*2 for i in List]  
Original\_List = [1, 2, 3, 4]  
print(pure\_func(Original\_List)) # Output: [1, 4, 9, 16]

2. \*\*Recursion\*\*:  
 - Functional programming avoids loops (`for`/`while`) and uses recursion for iteration.  
 - Example:

def Sum(L, i, n, count):  
 if n <= i:  
 return count  
 return Sum(L, i + 1, n, count + L[i])  
print(Sum([1, 2, 3, 4, 5], 0, 5, 0)) # Output: 15

3. \*\*First-Class and Higher-Order Functions\*\*:  
 - Functions are treated as first-class objects (can be passed as arguments, returned, or stored in variables).  
 - Example:

def shout(text): return text.upper()  
def greet(func): print(func("Hello"))  
greet(shout) # Output: HELLO

4. \*\*Immutability\*\*:  
 - Variables cannot be modified after initialization.  
 - Example:

immutable = "GeeksforGeeks"  
# immutable[1] = 'K' # Raises TypeError

---  
  
### \*\*Built-in Higher-Order Functions\*\*  
1. \*\*`map()`\*\*:  
 - Applies a function to all elements in an iterable.  
 - Example:

def addition(n): return n + n  
print(list(map(addition, [1, 2, 3, 4]))) # Output: [2, 4, 6, 8]

2. \*\*`filter()`\*\*:  
 - Filters elements based on a condition.  
 - Example:

def is\_vowel(char): return char in 'aeiou'  
print(list(filter(is\_vowel, 'geeksforgeeks'))) # Output: ['e', 'e', 'o', 'e', 'e']

3. \*\*Lambda Functions\*\*:  
 - Anonymous functions defined using the `lambda` keyword.  
 - Example:

cube = lambda x: x\*\*3  
print(cube(3)) # Output: 27

---

|  |  |
| --- | --- |
| **\*\*Functional Programming\*\*** | **\*\*Object-Oriented Programming\*\*** |
| Emphasizes functions and immutability. | Emphasizes objects and mutable data. |
| Declarative programming model. | Imperative programming model. |
| Uses recursion for iteration. | Uses loops for iteration. |
| Supports parallel programming. | Does not inherently support parallelism. |

---  
  
  
1. \*\*What is functional programming in Python?\*\*  
 - It emphasizes using functions, avoiding mutable data, and treating computation as mathematical function evaluation.  
  
2. \*\*What are common functional programming methods in Python?\*\*  
 - `map()`, `filter()`, `reduce()`, `lambda`, and list comprehensions.  
  
3. \*\*Why is Python considered functional?\*\*  
 - Python supports first-class functions, higher-order functions, and tools like `map()` and `filter()`.  
  
---  
  
This article provides a comprehensive overview of functional programming in Python, including its principles, examples, and built-in tools.

Lambda

### \*\*1. Basic Lambda Function\*\*  
A `lambda` function is an anonymous function defined using the `lambda` keyword. It can take any number of arguments but has only one expression.  
  
```python  
# Syntax: lambda arguments: expression  
square = lambda x: x \*\* 2  
print(square(5)) # Output: 25  
```  
  
---  
  
### \*\*2. Lambda with Multiple Arguments\*\*  
You can pass multiple arguments to a `lambda` function.  
  
```python  
# Lambda with two arguments  
add = lambda x, y: x + y  
print(add(3, 5)) # Output: 8  
```  
  
---  
  
### \*\*3. Using Lambda with `map()`\*\*  
The `map()` function applies a lambda function to each element in an iterable.  
  
```python  
numbers = [1, 2, 3, 4]  
squared = map(lambda x: x \*\* 2, numbers)  
print(list(squared)) # Output: [1, 4, 9, 16]  
```  
  
---  
  
### \*\*4. Using Lambda with `filter()`\*\*  
The `filter()` function filters elements in an iterable based on a condition defined by a lambda function.  
  
```python  
numbers = [1, 2, 3, 4, 5, 6]  
even\_numbers = filter(lambda x: x % 2 == 0, numbers)  
print(list(even\_numbers)) # Output: [2, 4, 6]  
```  
  
---  
  
### \*\*5. Using Lambda with `reduce()`\*\*  
The `reduce()` function (from the `functools` module) applies a lambda function cumulatively to the items in an iterable.  
  
```python  
from functools import reduce  
  
numbers = [1, 2, 3, 4]  
product = reduce(lambda x, y: x \* y, numbers)  
print(product) # Output: 24  
```  
  
---  
  
### \*\*6. Lambda Inside a Function\*\*  
You can define and use a `lambda` function inside another function.  
  
```python  
def multiplier(n):  
 return lambda x: x \* n  
  
double = multiplier(2)  
print(double(5)) # Output: 10  
```  
  
---  
  
### \*\*7. Sorting with Lambda\*\*  
You can use a `lambda` function as the key for sorting.  
  
```python  
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]  
# Sort by age (second element in the tuple)  
sorted\_students = sorted(students, key=lambda x: x[1])  
print(sorted\_students) # Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]  
```  
  
---  
  
### \*\*8. Lambda with Conditional Expressions\*\*  
You can use conditional logic in a `lambda` function.  
  
```python  
# Check if a number is even or odd  
check\_even\_odd = lambda x: "Even" if x % 2 == 0 else "Odd"  
print(check\_even\_odd(4)) # Output: Even  
print(check\_even\_odd(7)) # Output: Odd  
```  
  
---  
  
### \*\*9. Lambda with List Comprehension\*\*  
You can use a `lambda` function inside a list comprehension.  
  
```python  
numbers = [1, 2, 3, 4]  
squared = [(lambda x: x \*\* 2)(x) for x in numbers]  
print(squared) # Output: [1, 4, 9, 16]  
```  
  
---  
  
### \*\*10. Lambda for String Operations\*\*  
You can use `lambda` functions for string manipulations.  
  
```python  
# Convert a string to uppercase  
to\_upper = lambda s: s.upper()  
print(to\_upper("hello")) # Output: HELLO  
  
# Reverse a string  
reverse\_string = lambda s: s[::-1]  
print(reverse\_string("hello")) # Output: olleh  
```  
  
---  
  
### Summary  
`lambda` functions are concise and useful for short, throwaway functions, especially when used with higher-order functions like `map()`, `filter()`, and `reduce()`. However, for more complex logic, it's better to use regular `def` functions for readability.

List Comprehension

### \*\*1. Basic List Comprehension\*\*  
Create a new list by applying an operation to each element in an existing list.  
  
```python  
# Square each number in the list  
numbers = [1, 2, 3, 4, 5]  
squared = [x \*\* 2 for x in numbers]  
print(squared) # Output: [1, 4, 9, 16, 25]  
```  
  
---  
  
### \*\*2. List Comprehension with Conditional Filtering\*\*  
Include a condition to filter elements.  
  
```python  
# Get only even numbers from the list  
numbers = [1, 2, 3, 4, 5, 6]  
even\_numbers = [x for x in numbers if x % 2 == 0]  
print(even\_numbers) # Output: [2, 4, 6]  
```  
  
---  
  
### \*\*3. Nested Loops in List Comprehension\*\*  
Use nested loops to create combinations.  
  
```python  
# Create a list of all pairs (x, y) where x is from [1, 2] and y is from [3, 4]  
pairs = [(x, y) for x in [1, 2] for y in [3, 4]]  
print(pairs) # Output: [(1, 3), (1, 4), (2, 3), (2, 4)]  
```  
  
---  
  
### \*\*4. List Comprehension with `if-else`\*\*  
Include an `if-else` condition in the comprehension.  
  
```python  
# Replace even numbers with "Even" and odd numbers with "Odd"  
numbers = [1, 2, 3, 4, 5]  
result = ["Even" if x % 2 == 0 else "Odd" for x in numbers]  
print(result) # Output: ['Odd', 'Even', 'Odd', 'Even', 'Odd']  
```  
  
---  
  
### \*\*5. Flatten a Nested List\*\*  
Use list comprehension to flatten a 2D list into a 1D list.  
  
```python  
# Flatten a 2D list  
nested\_list = [[1, 2], [3, 4], [5, 6]]  
flattened = [x for sublist in nested\_list for x in sublist]  
print(flattened) # Output: [1, 2, 3, 4, 5, 6]  
```  
  
---  
  
### \*\*6. List Comprehension with Strings\*\*  
Perform operations on strings.  
  
```python  
# Convert each word to uppercase  
words = ["hello", "world", "python"]  
uppercase\_words = [word.upper() for word in words]  
print(uppercase\_words) # Output: ['HELLO', 'WORLD', 'PYTHON']  
```  
  
---  
  
### \*\*7. Generate a List of Numbers\*\*  
Use list comprehension to generate a sequence of numbers.  
  
```python  
# Generate a list of numbers from 0 to 9  
numbers = [x for x in range(10)]  
print(numbers) # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
```  
  
---  
  
### \*\*8. Cartesian Product\*\*  
Generate all combinations of elements from two lists.  
  
```python  
# Cartesian product of two lists  
list1 = [1, 2]  
list2 = ['a', 'b']  
cartesian\_product = [(x, y) for x in list1 for y in list2]  
print(cartesian\_product) # Output: [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]  
```  
  
---  
  
### \*\*9. Remove Vowels from a String\*\*  
Filter out vowels from a string using list comprehension.  
  
```python  
# Remove vowels from a string  
text = "geeksforgeeks"  
vowels = "aeiou"  
no\_vowels = ''.join([char for char in text if char not in vowels])  
print(no\_vowels) # Output: "gksfrgks"  
```  
  
---  
  
### \*\*10. Transpose a Matrix\*\*  
Use list comprehension to transpose a 2D matrix.  
  
```python  
# Transpose a matrix  
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]  
print(transposed) # Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]  
```  
  
---  
  
### \*\*11. Create a Dictionary from a List\*\*  
Use list comprehension to create a dictionary.  
  
```python  
# Create a dictionary with numbers as keys and their squares as values  
numbers = [1, 2, 3, 4]  
squared\_dict = {x: x \*\* 2 for x in numbers}  
print(squared\_dict) # Output: {1: 1, 2: 4, 3: 9, 4: 16}  
```  
  
---  
  
### \*\*12. Filter and Modify Simultaneously\*\*  
Filter elements and apply a transformation in one step.  
  
```python  
# Double the even numbers  
numbers = [1, 2, 3, 4, 5, 6]  
doubled\_evens = [x \* 2 for x in numbers if x % 2 == 0]  
print(doubled\_evens) # Output: [4, 8, 12]  
```  
  
---  
  
### \*\*13. Nested List Comprehension\*\*  
Use nested list comprehensions for more complex operations.  
  
```python  
# Create a multiplication table  
table = [[x \* y for y in range(1, 6)] for x in range(1, 6)]  
print(table)  
# Output: [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]  
```  
  
---  
  
### \*\*14. Filter Non-Numeric Values\*\*  
Filter out non-numeric values from a list.  
  
```python  
# Filter only integers from a mixed list  
mixed\_list = [1, 'a', 2, 'b', 3, 'c']  
numbers = [x for x in mixed\_list if isinstance(x, int)]  
print(numbers) # Output: [1, 2, 3]  
```  
  
---  
  
### \*\*15. Generate a List of Tuples\*\*  
Generate tuples using list comprehension.  
  
```python  
# Generate a list of (number, square) tuples  
numbers = [1, 2, 3, 4]  
tuples = [(x, x \*\* 2) for x in numbers]  
print(tuples) # Output: [(1, 1), (2, 4), (3, 9), (4, 16)]  
```  
  
---  
  
### Summary  
List comprehensions are a concise and powerful way to create and manipulate lists in Python. They are often more readable and efficient than traditional `for` loops.