

# 04\_Functional Programming

**Functional programming (FP)** is a programming paradigm focused on writing software by composing and applying **pure functions**, avoiding shared state, and minimizing side effects. It's particularly well-suited for **mathematical computations**, **data transformations**, and scenarios requiring **parallel processing**.

---

## Key Principles of Functional Programming

### 1. Pure Functions

- A **pure function** is a function where:
  - The output depends only on its inputs.
  - It has no side effects (doesn't modify external state).

**Example:**

```
# Pure function
def add(a, b):
    return a + b
```

- **Not Pure** (has side effects):

```
result = 0

def add(a, b):
    global result
    result = a + b
    return result
```

### Why It Matters:

- Easier to debug: The function's behavior is predictable and testable.
  - Parallelization: Pure functions can be executed independently.
- 

## 2. Immutability

- **Data is not modified** after it is created.
- Instead of changing data, new data structures are created.

**Example:**

```
# Immutable transformation
numbers = [1, 2, 3]
new_numbers = [x * 2 for x in numbers]
```

## Why It Matters:

- Reduces bugs caused by unexpected state changes.
  - Makes reasoning about program behavior easier.
- 

## 3. Higher-Order Functions

- Functions that take other functions as arguments or return functions.

**Example:**

```
# Map applies a function to each element
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared)) # Output: [1, 4, 9, 16]
```

## Why It Matters:

- Encourages reusability and modularity by composing small, reusable functions.
- 

## 4. First-Class Functions

- Functions are treated like data: They can be passed as arguments, returned from other functions, and assigned to variables.

**Example:**

```
def greet(name):  
    return f"Hello, {name}!"  
  
def execute(func, arg):  
    return func(arg)  
  
print(execute(greet, "Alice")) # Output: "Hello, Alice!"
```

## Why It Matters:

- Enables concise and expressive code.
- 

## 5. Recursion

- Instead of loops, functional programming often uses **recursion** to repeat operations.

**Example** (factorial with recursion):

```
def factorial(n):  
    return 1 if n == 0 else n * factorial(n - 1)  
  
print(factorial(5)) # Output: 120
```

## Why It Matters:

- Recursion avoids mutable state and aligns with the FP principle of immutability.
- 

## 6. Lazy Evaluation

- Computation is deferred until the result is actually needed.
- Common in FP languages like **Haskell**, but supported in Python via generators.

### Example:

```
# Lazy evaluation with a generator
def generate_numbers():
    for i in range(10):
        yield i

numbers = generate_numbers()
for num in numbers:
    print(num) # Generates each number one at a time
```

## Why It Matters:

- Optimizes performance by avoiding unnecessary computations.
  - Handles large or infinite data structures efficiently.
- 

## 7. Function Composition

- Combine smaller functions to build more complex functions.

### Example:

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

def square(x):
    return x ** 2

def compose(f, g):
    return lambda x: f(g(x))

double_then_square = compose(square, double)
print(double_then_square(3)) # Output: 36
```

## Why It Matters:

- Encourages modular and reusable code.

---

# Advantages of Functional Programming

1. **Predictable Code:**
    - Pure functions ensure the output is consistent, making debugging easier.
  2. **Concurrency and Parallelism:**
    - No shared state or side effects mean functions can run independently.
  3. **Modularity:**
    - Encourages writing small, reusable functions that can be combined in powerful ways.
  4. **Testability:**
    - Pure functions are easy to test as they don't depend on external states.
  5. **Immutable Data:**
    - Reduces bugs caused by unexpected changes to shared data.
- 

# Disadvantages of Functional Programming

1. **Learning Curve:**
    - The paradigm requires a shift in thinking for those used to procedural or object-oriented programming.
  2. **Performance:**
    - Immutable data structures can sometimes lead to higher memory usage and slower performance compared to mutable ones.
  3. **Debugging Recursion:**
    - Heavy reliance on recursion can lead to stack overflow errors if not optimized (e.g., via tail recursion).
  4. **Limited Libraries:**
    - Some libraries or APIs are built with OOP in mind and may not work well with FP.
-

# Functional Programming in Popular Languages

## Functional-First Languages:

- **Haskell:** Purely functional, lazy evaluation.
- **Erlang:** High concurrency and reliability.

## Functional Features in Multi-Paradigm Languages:

### 1. Python:

- Supports functional constructs like `map`, `filter`, `lambda`, and comprehensions.
- Example:

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x ** 2, nums))
print(squares) # Output: [1, 4, 9, 16]
```

### 2. JavaScript:

- Functional tools like `reduce`, `map`, and `filter`.
- Example:

```
const nums = [1, 2, 3, 4];
const squares = nums.map(x => x ** 2);
console.log(squares); // Output: [1, 4, 9, 16]
```

### 3. C++:

- Lambdas and standard functional algorithms in the STL (`std::transform`, `std::accumulate`).
- Example:

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
```

```
std::vector<int> nums = {1, 2, 3, 4};  
std::transform(nums.begin(), nums.end(), nums.begin(), [](int x) { return x * x; });  
for (int n : nums) std::cout << n << " "; // Output: 1 4 9 16  
}
```

---

# Applications of Functional Programming

## 1. Graphics Programming:

- Procedural texture generation and transformations (e.g., GLSL shaders).
- Functional paradigms simplify operations on immutable data like pixels or vertex buffers.

## 2. Data Processing:

- Big data frameworks like Apache Spark rely on FP for parallelism and immutability.

## 3. Game Development:

- Functional constructs help build procedural systems like terrain generation or AI logic.

## 4. Concurrency:

- Functional programming is ideal for writing highly concurrent and parallel systems due to immutability.

---

Would you like more hands-on examples in Python or another language, or a deeper dive into functional constructs? ☐

---

Revision #1

Created 29 December 2024 22:39:52 by victor

Updated 29 December 2024 22:40:56 by victor