LECTURE 7. PART I. Functional programming

1. Basic principles

In object-oriented programming (OOP), you create "objects" (hence the name), which are structures that have data and methods. In functional programming, everything is a function. Functional programming tries to keep data and behavior separate, and OOP brings those concepts together.

There are two main things needed to be known to understand the concept:

- **Data is immutable**: If you want to change data, such as an array, you return a new array with the changes, not the original.
- **Functions are stateless**: Functions act as if for the first time, every single time! In other words, the function always gives the same return value for the same arguments.

Functional programming is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

1.1. Pure functions

The first fundamental concept we learn when we want to understand functional programming is pure functions. But what does that really mean? What makes a function pure? How do we know if a function is pure or not? Here is a very strict definition of purity:

- It returns the same result if given the same arguments (it is also referred as deterministic)
- It does not cause any observable side effects

Remarks:

- A function whose result depends on other values than the function parameters is not pure. Such a call may return different results for different calls with same parameters.
- A function reading external files is not a pure function, as the file's contents may change.
- A function depending on a random number generator is not a pure function.
- Modifying a global object is discouraged, as it leads to impure functions.

1.2. Immutability

When data is immutable, its state cannot change after it's created. If you want to change an immutable object, you can't. Instead, you create a new object with the new value. With recursion, we keep the variables immutable.

1.3. Referential transparency

Let's implement a *square* function. This (pure) function will always have the same output, given the same input. Passing "2" as a parameter of the *square* function will always return 4. So now we can replace the *(square 2)* with *4*. Basically, if a function consistently yields the same result for the same input, it is referentially transparent.

pure functions + immutable data = referential transparency

1.4. Functions as first-class entities

The idea of functions as first-class entities is that functions are also treated as values and used as data. Functions as first-class entities can:

- refer to it from constants and variables
- pass it as a parameter to other functions
- return it as result from other functions

The idea is to treat functions as values and pass functions like data. This way we can combine different functions to create new functions with new behavior.

1.5. Higher-order functions

When we talk about higher-order functions, we mean a function that either:

- takes one or more functions as arguments, or
- returns a function as its result

For example:

- Filter – Given a collection, filter by an attribute according to a logical test
- Map – Transform a collection by applying a function to all its elements
- Reduce – Given a function and a collection, return a value created from the items

2. Comparison

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups, i.e. −

- **Pure Functional Languages −** These types of functional languages support only the functional paradigms. For example − Haskell.

- **Impure Functional Languages −** These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

2.1. Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows −

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.

- Functional programming supports higher-order functions and lazy evaluation features.

- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.

2.2. Functional Programming – Advantages

Functional programming offers the following advantages −

- **Bugs-Free Code** − Functional programming does not support state, so there are no side-effect results and we can write error-free codes.

- **Efficient Parallel Programming** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.

- **Efficiency** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.

- **Supports Nested Functions** − Functional programming supports Nested Functions.

- **Lazy Evaluation** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.

Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.

- Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.

- Embedded Lisp interpreters add programmability to some systems like Emacs.

2.3. Functional Programming vs. Object-oriented Programming

The following highlights the major differences between functional programming and object-oriented programming −

Functional Programming	**Object-Oriented Programming**
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus on: "What you are doing"	Focus on "How you are doing"
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects

Functions are first-class citizens

Objects are first-class citizens

Flow control: function calls, function calls with recursion

Flow control: loops, conditional statements

Uses "Recursion" concept to iterate

Uses "Loop" concept to iterate

Execution order not so important

Execution order is very important

Mostly about "Abstraction over Behavior"

Mostly about "Abstraction over Data"

Use FP when we have few things with more operations.
Use OOP when we have few operations with more things.