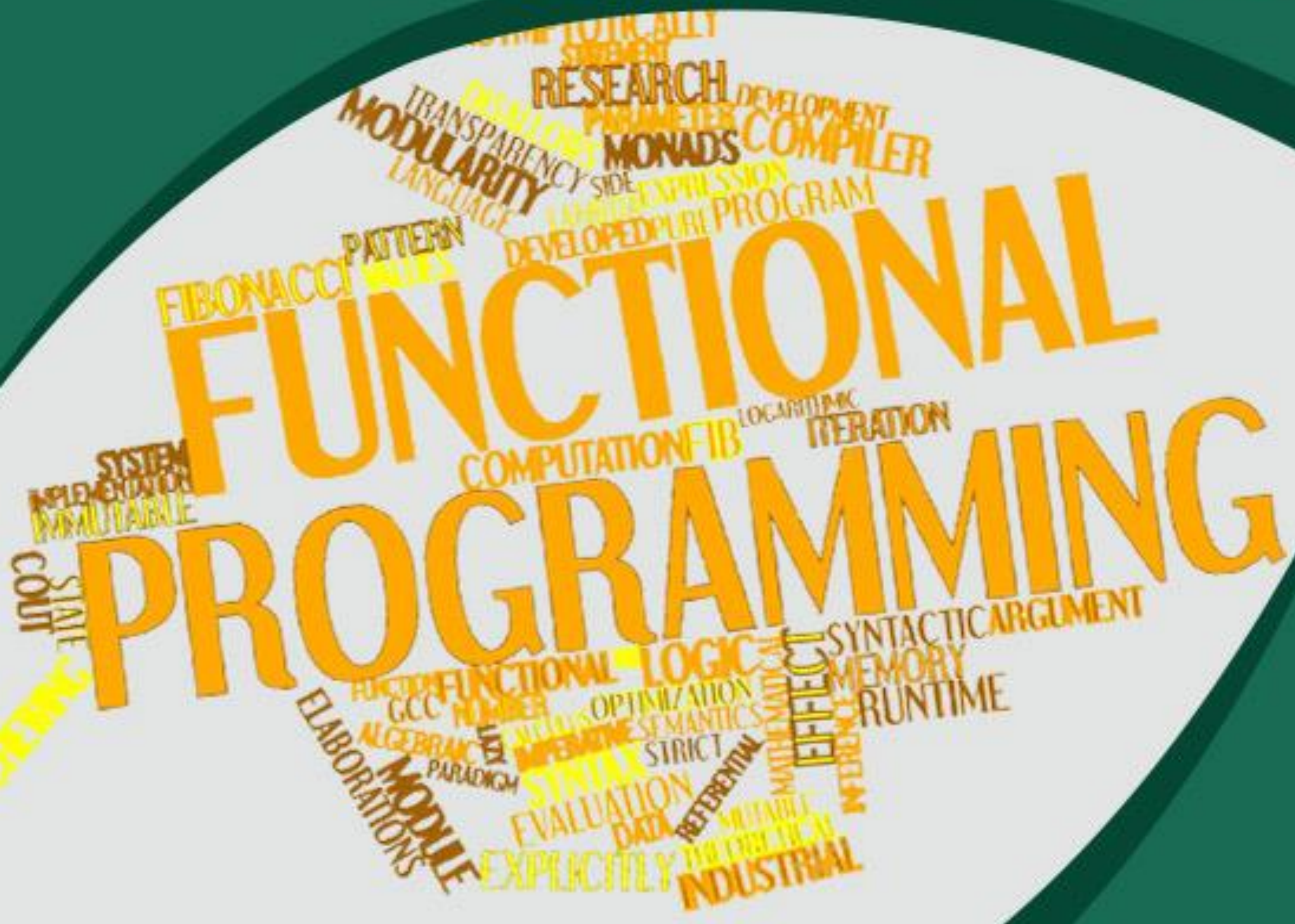

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

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.

This tutorial provides a brief overview of the most fundamental concepts of functional programming languages in general. In addition, it provides a comparative analysis of object-oriented programming and functional programming language in every example.

Audience

This tutorial will help all those readers who are keen to understand the basic concepts of functional programming. It is a very basic tutorial that has been designed keeping in mind the requirements of beginners.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of Computer Programming terminologies in general and a good exposure to any programming language such as C, C++, or Java.

Copyright & Disclaimer

Table of Contents

1. Functional Programming – Introduction

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.

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.

- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

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.

Functional Programming vs. Object-oriented Programming

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

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is 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.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

Efficiency of a Program Code

The efficiency of a programming code is directly proportional to the algorithmic efficiency and the execution speed. Good efficiency ensures higher performance.

The factors that affect the efficiency of a program includes:

- The speed of the machine
- Compiler speed
- Operating system
- Choosing right Programming language
- The way of data in a program is organized
- Algorithm used to solve the problem

The efficiency of a programming language can be improved by performing the following tasks:

- By removing unnecessary code or the code that goes to redundant processing.
- By making use of optimal memory and nonvolatile storage
- By making the use of reusable components wherever applicable.
- By making the use of error & exception handling at all layers of program.
- By creating programming code that ensures data integrity and consistency.
- By developing the program code that's compliant with the design logic and flow.

An efficient programming code can reduce resource consumption and completion time as much as possible with minimum risk to the operating environment.

2. Functional Programming – Functions Overview

In programming terms, a **function** is a block of statements that performs a specific task. Functions accept data, process it, and return a result. Functions are written primarily to support the concept of reusability. Once a function is written, it can be called easily, without having to write the same code again and again.

Different functional languages use different syntax to write a function.

Prerequisites to Writing a Function

Before writing a function, a programmer must know the following points:

- Purpose of function should be known to the programmer.
- Algorithm of the function should be known to the programmer.
- Functions data variables & their goal should be known to the programmer.
- Function's data should be known to the programmer that is called by the user.

Flow Control of a Function

When a function is "called", the program "transfers" the control to execute the function and its "flow of control" is as below:

- The program reaches to the statement containing a "function call".
- The first line inside the function is executed.
- All the statements inside the function are executed from top to bottom.
- When the function is executed successfully, the control goes back to the statement where it started from.
- Any data computed and returned by the function is used in place of the function in the original line of code.

Syntax of a Function

The general syntax of a function looks as follows:

```
returnType functionName(type1 argument1, type2 argument2, . . . )
{
     // function body
}
```

Defining a Function in C++

Let’s take an example to understand how a function can be defined in C++ which is an object-oriented programming language. The following code has a function that adds two numbers and provides its result as the output.

```
#include <stdio.h>
int addNum(int a, int b);             // function prototype

int main()
{
    int sum;
    sum = addNum(5,6);                // function call
    printf("sum = %d",sum);
    return 0;
}

int addNum (int a,int b)              // function definition
{
    int result;
    result = a+b;
    return result;                    // return statement
}
```

It will produce the following output:

```
Sum=11
```

Defining a Function in Erlang

Let’s see how the same function can be defined in Erlang, which is a functional programming language.

```
-module(helloworld).
-export([add/2,start/0]).

add(A,B) ->
```

```
   C = A+B,
   io:fwrite("~w~n",[C]).
start() ->
   add(5,6).
```

It will produce the following output:

```
11
```

Function Prototype

A function prototype is a declaration of the function that includes return-type, function-name & arguments-list. It is similar to function definition without function-body.

For Example: Some programming languages supports function prototyping & some are not.

In C++, we can make function prototype of function 'sum' like this:

```
int sum(int a, int b)
```

Note: Programming languages like Python, Erlang, etc doesn't supports function prototyping, we need to declare the complete function.

What is the use of function prototype?

The function prototype is used by the compiler when the function is called. Compiler uses it to ensure correct return-type, proper arguments list are passed-in, & their return-type is correct.

Function Signature

A function signature is similar to function prototype in which number of parameters, data-type of parameters & order of appearance should be in similar order. For Example:

```
void Sum(int a, int b, int c);            // function 1


void Sum(float a, float b, float c);        // function 2


void Sum(float a, float b, float c);        // function 3
```

Function1 and Function2 have different signatures. Function2 and Function3 have same signatures.

Note: Function overloading and Function overriding which we will discuss in the subsequent chapters are based on the concept of function signatures.

- Function overloading is possible when a class has multiple functions with the same name but different signatures.

- Function overriding is possible when a derived class function has the same name and signature as its base class.

3. Functional Programming – Function Types

Functions are of two types:

- Predefined functions
- User-defined functions

In this chapter, we will discuss in detail about functions.

Predefined Functions

These are the functions that are built into Language to perform operations & are stored in the Standard Function Library.

For Example: 'Strcat' in C++ & 'concat' in Haskell are used to append the two strings, 'strlen' in C++ & 'len' in Python are used to calculate the string length.

Program to print string length in C++

The following program shows how you can print the length of a string using C++:

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;

int main()
{
     char str[20]="Hello World";
     int len;
     len=strlen(str);
     cout<<"String length is: "<<len;
     return 0;
}
```

It will produce the following output:

```
String length is: 11
```

Program to print string length in Python

The following program shows how to print the length of a string using Python, which is a functional programming language:

```
str = "Hello World";

print("String length is: ", len(str))
```

It will produce the following output:

```
String length is: 11
```

User-defined Functions

User-defined functions are defined by the user to perform specific tasks. There are four different patterns to define a function:

- Functions with no argument and no return value
- Functions with no argument but a return value
- Functions with argument but no return value
- Functions with argument and a return value

Functions with no argument and no return value

The following program shows how to define a function with no argument and no return value **in C++**:

```
#include <iostream>

using namespace std;

void function1()

{

    cout <<"Hello World";

}


int main()

{

     function1();

     return 0;

}
```

It will produce the following output:

```
Hello World
```

The following program shows how you can define a similar function (no argument and no return value) **in Python**:

```
def function1():
   print ("Hello World")


function1()
```

It will produce the following output:

```
Hello World
```

Functions with no argument but a return value

The following program shows how to define a function with no argument but a return value **in C++**:

```
#include <iostream>
using namespace std;
string function1()
{
    return("Hello World");
}

int main()
{
     cout<<function1();
     return 0;
}
```

It will produce the following output:

```
Hello World
```

The following program shows how you can define a similar function (with no argument but a return value) **in Python**:

```
def function1():
    return "Hello World"
res = function1()
print(res)
```

It will produce the following output:

```
Hello World
```

Functions with argument but no return value

The following program shows how to define a function with argument but no return value **in C++**:

```
#include <iostream>
using namespace std;
void function1(int x, int y)
{
     int c;
     c=x+y;
     cout<<"Sum is: "<<c;
}

int main()
{
     function1(4,5);
     return 0;
}
```

It will produce the following output:

```
Sum is: 9
```

The following program shows how you can define a similar function **in Python**:

```
def function1(x,y):
    c = x + y
    print("Sum is:",c)
function1(4,5)
```

It will produce the following output:

```
Sum is: 9
```

Functions with argument and a return value

The following program shows how to define a function **in C++** with no argument but a return value:

```
#include <iostream>
using namespace std;
int function1(int x, int y)
{
     int c;
     c=x+y;
     return c;
}
int main()
{
     int res;
     res=function1(4,5);
     cout<<"Sum is: "<<res;
     return 0;
}
```

It will produce the following output:

```
Sum is: 9
```

The following program shows how to define a similar function (with argument and a return value) **in Python**:

```
def function1(x,y):
    c = x + y
    return c

res = function1(4,5)
print("Sum is ",res)
```

It will produce the following output:

```
Sum is 9
```