# F# - FUNCTIONS

 

In F#, functions work like data types. You can declare and use a function in the same way like any other variable.

Since functions can be used like any other variables, you can −

- Create a function, with a name and associate that name with a type.
- Assign it a value.
- Perform some calculation on that value.
- Pass it as a parameter to another function or sub-routine.
- Return a function as the result of another function.

## Defining a Function

Functions are defined by using the **let** keyword. A function definition has the following syntax −

```
let [inline] function-name parameter-list [ : return-type ]
= function-body
```

Where,

- **function-name** is an identifier that represents the function.
- **parameter-list** gives the list of parameters separated by spaces. You can also specify an explicit type for each parameter and if not specified compiler tends to deduce it from the function body *likevariables*.
- **function-body** consists of an expression, or a compound expression consisting of a number of expressions. The final expression in the function body is the return value.
- **return-type** is a colon followed by a type and is optional. If the return type is not specified, then the compiler determines it from the final expression in the function body.

## Parameters of a Function

You list the names of parameters right after the function name. You can specify the type of a parameter. The type of the parameter should follow the name of the parameter separated by a colon.

If no parameter type is specified, it is inferred by the compiler.

For example −

```
let doubleIt (x : int) = 2 * x
```

## Calling a Function

A function is called by specifying the function name followed by a space and then any arguments separated by spaces.

For example −

```
let vol = cylinderVolume 3.0 5.0
```

The following programs illustrate the concepts.

## Example 1

The following program calculates the volume of a cylinder when the radius and length are given as parameters.

```
// the function calculates the volume of
// a cylinder with radius and length as parameters

let cylinderVolume radius length : float =

   // function body
   let pi = 3.14159
   length * pi * radius * radius

let vol = cylinderVolume 3.0 5.0
printfn " Volume: %g " vol
```

When you compile and execute the program, it yields the following output −

```
Volume: 141.372
```

## Example 2

The following program returns the larger value of two given parameters −

```
// the function returns the larger value between two
// arguments

let max num1 num2 : int32 =
   // function body
   if(num1>num2)then
      num1
   else
      num2

let res = max 39 52
printfn " Max Value: %d " res
```

When you compile and execute the program, it yields the following output −

```
Max Value: 52
```

## Example 3

```
let doubleIt (x : int) = 2 * x
printfn "Double 19: %d" ( doubleIt(19))
```

When you compile and execute the program, it yields the following output −

```
Double 19: 38
```

## Recursive Functions

Recursive functions are functions that call themselves.

You define a recursive using the **let rec** keyword combination.

Syntax for defining a recursive function is −

```
//Recursive function definition
let rec function-name parameter-list = recursive-function-body
```

For example −

```
let rec fib n = if n < 2 then 1 else fib (n - 1) &plus; fib (n - 2)
```

## Example 1

The following program returns Fibonacci 1 to 10 −

```
let rec fib n = if n < 2 then 1 else fib (n - 1) &plus; fib (n - 2)
for i = 1 to 10 do
   printfn "Fibonacci %d: %d" i (fib i)
```

When you compile and execute the program, it yields the following output −

```
Fibonacci 1: 1
Fibonacci 2: 2
Fibonacci 3: 3
Fibonacci 4: 5
Fibonacci 5: 8
Fibonacci 6: 13
Fibonacci 7: 21
Fibonacci 8: 34
Fibonacci 9: 55
Fibonacci 10: 89
```

## Example 2

The following program returns factorial 8 −

```
open System
let rec fact x =
   if x < 1 then 1
   else x * fact (x - 1)
Console.WriteLine(fact 8)
```

When you compile and execute the program, it yields the following output −

```
40320
```

## Arrow Notations in F#

F# reports about data type in functions and values, using a chained arrow notation. Let us take an example of a function that takes one *int* input, and returns a string. In arrow notation, it is written as −

```
int -> string
```

Data types are read from left to right.

Let us take another hypothetical function that takes two int data inputs and returns a string.

```
let mydivfunction x y = (x / y).ToString();;
```

F# reports the data type using chained arrow notation as −

```
val mydivfunction : x:int -> y:int -> string
```

The return type is represented by the rightmost data type in chained arrow notation.

Some more examples −

| Notation | Meaning |
| --- | --- |

| | |
|---|---|
| float → float → float | The function takes two *float* inputs, returns another *float*. |
| int → string → float | The function takes an *int* and a *string* input, returns a *float*. |

## Lambda Expressions

A **lambda expression** is an unnamed function.

Let us take an example of two functions −

```
let applyFunction ( f: int -> int -> int) x y = f x y
let mul x y = x * y
let res = applyFunction mul 5 7
printfn "%d" res
```

When you compile and execute the program, it yields the following output −

```
35
```

Now in the above example, if instead of defining the function *mul,* we could have used lambda expressions as −

```
let applyFunction ( f: int -> int -> int) x y = f x y
let res = applyFunction (fun x y -> x * y ) 5 7
printfn "%d" res
```

When you compile and execute the program, it yields the following output −

```
35
```

## Function Composition and Pipelining

In F#, one function can be composed from other functions.

The following example shows the composition of a function named f, from two functions function1 and function2 −

```
let function1 x = x + 1
let function2 x = x * 5

let f = function1 >> function2
let res = f 10
printfn "%d" res
```

When you compile and execute the program, it yields the following output −

```
55
```

F# also provides a feature called **pipelining of functions.** Pipelining allows function calls to be chained together as successive operations.

The following example shows that −

```
let function1 x = x + 1
let function2 x = x * 5

let res = 10 |> function1 |> function2
printfn "%d" res
```

When you compile and execute the program, it yields the following output −

```
55
```