Unit I

CHAPTER 1

FUNCTION

3FY1S

Learning Objectives

After the completion of this chapter, the student will be able to:

- Understand Function Specification.
- Parameters (and arguments).
- Interface Vs Implementation.
- Pure functions.
- Side - effects (impure functions).

1.1 Introduction

The most important criteria in writing and evaluating the algorithm is the time it takes to complete a task. To have a meaningful comparison of algorithms, the duration of computation time must be independent of the programming language, compiler, and computer used. As you aware that algorithms are expressed using statements of a programming language. If a bulk of statements to be repeated for many numbers of times then subroutines are used to finish the task.

Subroutines are the basic building blocks of computer programs. Subroutines are small sections of code that are used to perform a particular task that can be used repeatedly. In Programming languages these subroutines are called as Functions.

1.2 Function with respect to Programming language

A function is a unit of code that is often defined within a greater code structure. Specifically, a function contains a set of code that works on many kinds of inputs, like variants, expressions and produces a concrete output.

1.2.1 Function Specification

Let us consider the example *a:= (24).* *a:= (24)* has an expression in it but (24) is not itself an expression. Rather, it is a function definition. Definitions bind values to names, in this case the value 24 being bound to the name *'a'*. Definitions are not expressions, at the same time expressions are also not treated as definitions. Definitions are distinct syntactic blocks. Definitions can have expressions nested inside them, and vice-versa.

1.2.2 Parameters (and arguments)

Parameters are the variables in a function definition and arguments are the values which are passed to a function definition.

1. Parameter without Type

Let us see an example of a function definition:

```
(requires: b>=0 )
(returns: a to the power of b)
let rec pow a b:=
        if b=0 then 1
        else a * pow a (b-1)
```

In the above function definition variable *'b'* is the parameter and the value which is passed to the variable *'b'* is the argument. The precondition ***(requires)*** and postcondition ***(returns)*** of the function is given. Note we have not mentioned any types: ***(data types)***. Some language compiler solves this type ***(data type)*** inference problem algorithmically, but some require the type to be mentioned.

In the above function definition if expression can return ***1*** in the then branch, shows that as per the ***typing*** rule the entire if expression has type ***int***. Since the if expression is of type ***'int'***, the function's return type also be *'int'*. ***'b'*** is compared to *0* with the equality operator, so ***'b'*** is also a type of *'int'*. Since ***'a'*** is multiplied with another expression using the * operator, ***'a'*** must be an int.

2. Parameter with Type

Now let us write the same function definition with types for some reason:

```
(requires: b>=0 )
(returns: a to the power of b )
let rec pow (a: int) (b: int) : int :=
        if b=0 then 1
        else a * pow a (b-1)
```

When we write the type annotations for ***'a'*** and ***'b'*** the parentheses are mandatory. Generally we can leave out these annotations, because it's simpler to let the compiler infer them. There are times we may want to explicitly write down types. This is useful on times when you get a type error from the compiler that doesn't make sense. Explicitly annotating the types can help with debugging such an error message.

The syntax to define functions is close to the mathematical usage: the definition is introduced by the keyword ***let***, followed by the ***name*** of the function and its *arguments*; then the formula that computes the image of the argument is written after an := sign. If you want to define a recursive function: use ***"let rec" instead of "let".***

Syntax: The syntax for function definitions:

```
let rec fn a1 a2 ... an := k
```

Here the ***'fn'*** is a variable indicating an identifier being used as a function name. The names ***'a1'*** to ***'an'*** are variables indicating the identifiers used as parameters. The keyword ***'rec'*** is required if ***'fn'*** is to be a recursive function; otherwise it may be omitted.

Note

A function definition which call itself is called recursive function.

For example: let us see an example to check whether the entered number is even or odd.

```
(requires: x>= 0)
        let rec even x :=
        x=0 || odd (x-1)
        return 'even'
(requires: x>= 0)
        let odd x :=
        x<>0 && even (x-1)
        return 'odd'
```

The syntax for function types:

```
x → y
x1 → x2 → y
x1 → ... → xn → y
```

The *'x'* and *'y'* are variables indicating types. ***The type x→ y is the type of a function that gets an input of type 'x' and returns an output of type 'y'.*** Whereas ***x1 → x2 → y*** is a type of a function that takes two inputs, the first input is of type ***'x1'*** and the second input of type ***'x2'***, and returns an output of type ***'y'***. Likewise ***x1 → ... → xn → y*** has type ***'x'*** as input of n arguments and ***'y'*** type as output.

Note

All functions are static definitions. There is no dynamic function definitions.

1.3 Interface Vs Implementation

An interface is a set of action that an object can do. For example when you press a light switch, the light goes on, you may not have cared how it splashed the light. In Object Oriented Programming language, an Interface is a description of all functions that a class must have in order to be a new interface. In our example, anything that ***"ACTS LIKE"*** a light, should have function definitions like turn_on () and a turn_off (). The purpose of interface is to allow the computer to enforce the properties of the class of ***TYPE T*** *(whatever the interface is)* must have functions called *X, Y, Z*, etc.

A class declaration combines the external interface *(its local state)* with an implementation of that interface *(the code that carries out the behaviour)*. An object is an instance created from the class.

The interface defines an object's visibility to the outside world.

The difference between interface and implementation is

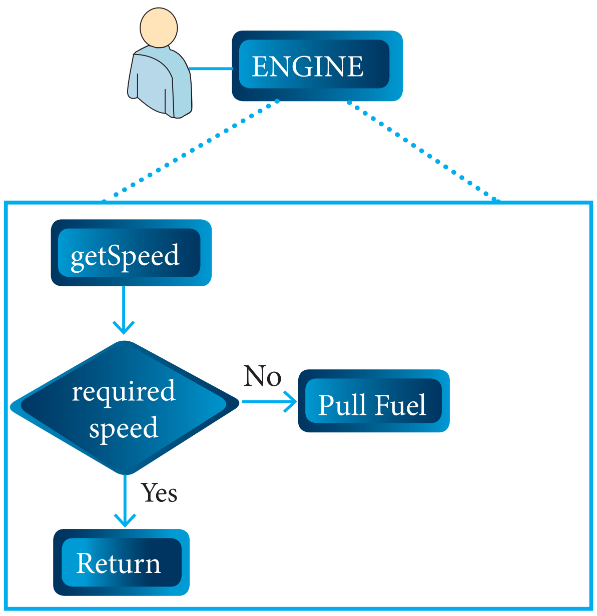
Interface	Implementation
Interface just defines what an object can do, but won't actually do it	Implementation carries out the instructions defined in the interface

In object oriented programs classes are the interface and how the object is processed and executed is the implementation.

1.3.1 Characteristics of interface

- The class template specifies the interfaces to enable an object to be created and operated properly.
- An object's attributes and behaviour is controlled by sending functions to the object.

For example, let's take the example of increasing a car's speed.



The person who drives the car doesn't care about the internal working. To increase the speed of the car he just presses the accelerator to get the desired behaviour. Here the accelerator is the interface between the driver *(the calling / invoking object)* and the engine *(the called object)*.

In this case, the function call would be Speed (70): This is the interface.

Internally, the engine of the car is doing all the things. It's where fuel, air, pressure, and electricity come together to create the power to move the vehicle. All of these actions are separated from the driver, who just wants to go faster. Thus we separate interface from implementation.

Let us see a simple example, consider the following implementation of a function that finds the minimum of its three arguments:

```
let min 3 x y z :=
        if x < y then
                if x < z then x else z
        else
                if y < z then y else z
```

1.4 Pure functions

Pure functions are functions which will give exact result when the same arguments are passed. For example the mathematical function sin (0) always results **0**. This means that every time you call the function with the same arguments, you will always get the same result. A function can be a pure function provided it should not have any external variable which will alter the behaviour of that variable.

Let us see an example

```
let square x
        return: x * x
```

The above function square is a pure function because it will not give different results for same input.

There are various theoretical advantages of having pure functions. One advantage is that if a function is pure, then if it is called several times with the same arguments, the compiler only needs to actually call the function once. Lt's see an example

```
let length s:=
        i: = 0
        if i <strlen (s) then
        -- Do something which doesn't affect s
        ++i
```

If it is compiled, ***strlen (s)*** is called each time and strlen needs to iterate over the whole of **'s'.** If the compiler is smart enough to work out that strlen is a pure function and that **'s'** is not updated in the loop, then it can remove the redundant extra calls to strlen and make the loop to execute only one time. From these what we can understand, strlen is a pure function because the function takes one variable as a parameter, and accesses it to find its length. This function reads external memory but does not change it, and the value returned derives from the external memory accessed.

Note

Evaluation of pure functions does not cause any side effects to its output

1.4.1 Impure functions

The variables used inside the function may cause side effects though the functions which are not passed with any arguments. In such cases the function is called impure function. When a function depends on variables or functions outside of its definition block, you can never be sure that the function will behave the same every time it's called. For example the mathematical function random() will give different outputs for the same function call.

```
let randomnumber :=
    a := random()
    if a > 10 then
        return: a
    else
        return: 10
```

Here the function Random is impure as it is not sure what will be the result when we call the function.

1.4.2 Side-effects (Impure functions)

As you are aware function has side effects when it has observable interaction with the outside world. There are situations our functions can become impure though our goal is to make our functions pure. Just to clarify remember that side effect is not a necessary bad thing.Sometimes they are useful *(especially outside functional programming paradigm)*.

Modify variable outside a function

One of the most popular groups of side effects is modifying the variable outside of function.

For example

```
let y: = 0
(int) inc (int) x:
    y: = y + x
    return (y)
```

In the above example the value of y get changed inside the function definition due to which the result will change each time. The side effect of the inc () function is it is changing the data of the external visible variable **'y'**. As you can see some side effects are quite easy to spot and some of them may tricky. A good sign that our function impure *(has side effect)* is that it doesn't take any arguments and it doesn't return any value.

From all these examples and definitions what we can understand about the main differences between pure and impure functions are

Pure Function	Impure Function
The return value of the pure functions solely depends on its arguments passed. Hence, if you call the pure functions with the same set of arguments, you will always get the same return values. They do not have any side effects.	The return value of the impure functions does not solely depend on its arguments passed. Hence, if you call the impure functions with the same set of arguments, you might get the different return values For example, random(), Date().
They do not modify the arguments which are passed to them	They may modify the arguments which are passed to them

Now let's see the example of a pure function to determine the greatest common divisor *(gcd)* of two positive integer numbers.

```
let rec gcd a b :=
    if b <> 0 then gcd b (a mod b)
    else
    return a
output
    gcd 13 27;
    1
    gcd 20536 7826
    2
```

In the above example program *'gcd'* is the name of the function which recursively called till the variable *'b'* becomes *'0'*. Remember *b* and *(a mod b)* are two arguments passed to *'a'* and *'b'* of the gcd function.

1.4.3 Chameleons of Chromeland problem using function

Recall the In the Chameleons of Chromeland problem what you have studied in class XI. suppose two types of chameleons are equal in number. Construct an algorithm that arranges meetings between these two types so that they change their color to the third type. In the end, all should display the same color.

Let us represent the number of chameleons of each type by variables *a, b* and *c*, and their initial values by *A, B* and *C*, respectively. Let a = b be the input property.

The input – output relation is a = b = 0 and c = A + B + C. Let us name the algorithm monochromatize. The algorithm can be specified as

monochromatize (a, b, c)

-- inputs	:	a = A, b = B, c = C, a = b
-- outputs	:	a = b = 0, c = A+B+C

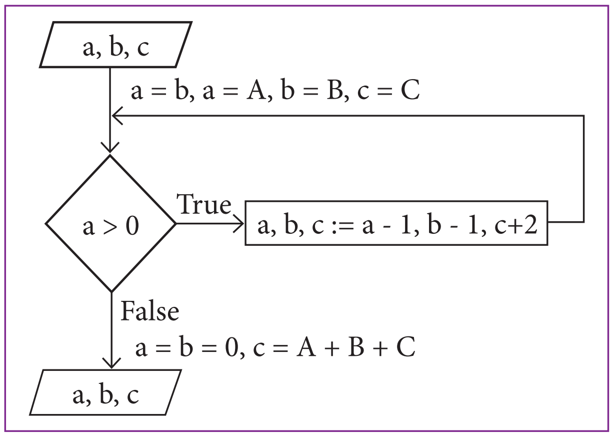
In each iterative step, two chameleons of the two types *(equal in number)* meet and change their colors to the third one. For example, if A, B, C = 4, 4, 6, then the series of meeting will result in

iteration	a	b	c
0	4	4	4
1	3	3	8
2	2	2	10
3	1	1	12
4	0	0	14

In each meeting, ***a*** and ***b*** each decreases by ***1***, and ***c*** increases by ***2***. The solution can be expressed as an iterative algorithm.

```
monochromatize (a, b,  c)
    -- inputs : a = A, b=B, c=C, a=b
    -- outputs : a = b = 0, c = A+B+C
while a>0
      a, b, c := a-1, b-1, c+2
```

The algorithm is depicted in the flowchart as below



Now let us write this algorithm using function

```
let rec monochromatize a  b  c :=
      if a > 0 then
        a, b, c := a-1, b-1, c+2
      else
        a:=0, b:=0, c:= a + b + c
        return c
```

Points to remember:

- Algorithms are expressed using statements of a programming language
- Subroutines are small sections of code that are used to perform a particular task that can be used repeatedly
- A function is a unit of code that is often defined within a greater code structure
- A function contains a set of code that works on many kinds of inputs and produces a concrete output
- Definitions are distinct syntactic blocks
- Parameters are the variables in a function definition and arguments are the values which are passed to a function definition through the function definition.
- When you write the type annotations the parentheses are mandatory in the function definition
- An interface is a set of action that an object can do
- Interface just defines what an object can do, but won't actually do it
- Implementation carries out the instructions defined in the interface
- Pure functions are functions which will give exact result when the same arguments are passed
- The variables used inside the function may cause side effects though the functions which are not passed with any arguments. In such cases the function is called impure function

Hands on Practice

1. Write algorithmic function definition to find the minimum among 3 numbers.
2. Write algorithmic recursive function definition to find the sum of n natural numbers.

S2LUO

Evaluation

Part - I

Choose the best answer **(1 Mark)**

1. The small sections of code that are used to perform a particular task is called

   (A) Subroutines (B) Files (C) Pseudo code (D) Modules

2. Which of the following is a unit of code that is often defined within a greater code structure?

   (A) Subroutines (B) Function (C) Files (D) Modules

3. Which of the following is a distinct syntactic block?

   (A) Subroutines (B) Function (C) Definition (D) Modules

4. The variables in a function definition are called as

   (A) Subroutines (B) Function (C) Definition (D) Parameters

5. The values which are passed to a function definition are called

   (A) Arguments (B) Subroutines (C) Function (D) Definition

6. Which of the following are mandatory to write the type annotations in the function definition?

   (A) Curly braces (B) Parentheses (C) Square brackets (D) indentations

7. Which of the following defines what an object can do?

   (A) Operating System (B) Compiler (C) Interface (D) Interpreter

8. Which of the following carries out the instructions defined in the interface?

   (A) Operating System (B) Compiler (C) Implementation (D) Interpreter

9. The functions which will give exact result when same arguments are passed are called

   (A) Impure functions (B) Partial Functions

   (C) Dynamic Functions (D) Pure functions

10. The functions which cause side effects to the arguments passed are called

(A) impure function (B) Partial Functions

(C) Dynamic Functions (D) Pure functions

Part - II

Answer the following questions **(2 Marks)**

1. What is a subroutine?
2. Define Function with respect to Programming language.
3. Write the inference you get from X:=(78).
4. Differentiate interface and implementation.
5. Which of the following is a normal function definition and which is recursive function definition

```
i)  let sum x y:
          return x + y
ii) let disp :
          print 'welcome'
iii) let rec sum num:
          if (num!=0) then return num + sum (num-1)
          else
          return num
```

Part - III

Answer the following questions **(3 Marks)**

1. Mention the characteristics of Interface.
2. Why strlen is called pure function?
3. What is the side effect of impure function. Give example.
4. Differentiate pure and impure function.
5. Wha happens if you modify a variable outside the function? Give an example.

Part - IV

Answer the following questions **(5Marks)**

1. What are called Parameters and write a note on

   (i) Parameter without Type (ii) Parameter with Type

2. Identify in the following program

```
let rec gcd a b :=
        if b <> 0 then gcd b (a mod b) else return a
```

   i) Name of the function

   ii) Identify the statement which tells it is a recursive function

   iii) Name of the argument variable

   iv) Statement which invoke the function recursively

   v) Statement which terminates the recursion

3. Explain with example Pure and impure functions.
4. Explain with an example interface and implementation.

REFERENCES

1. *Data Structures and Algorithms in Python By Michael T.Goodrich, RobertoTamassia and Michael H. Goldwasser.*
2. *Data Structure and Algorithmic Thinking in Python By Narasimha Karumanchi*
3. *https://www.python.org*