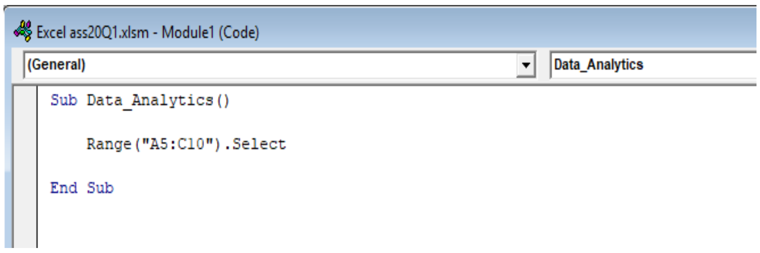
# ADVANCE EXCEL ASSIGNMENT 20

1. **Write a VBA code to select the cells from A5 to C10. Give it a name "Data Analytics" and fill the cells with the following "This is Excel VBA".**

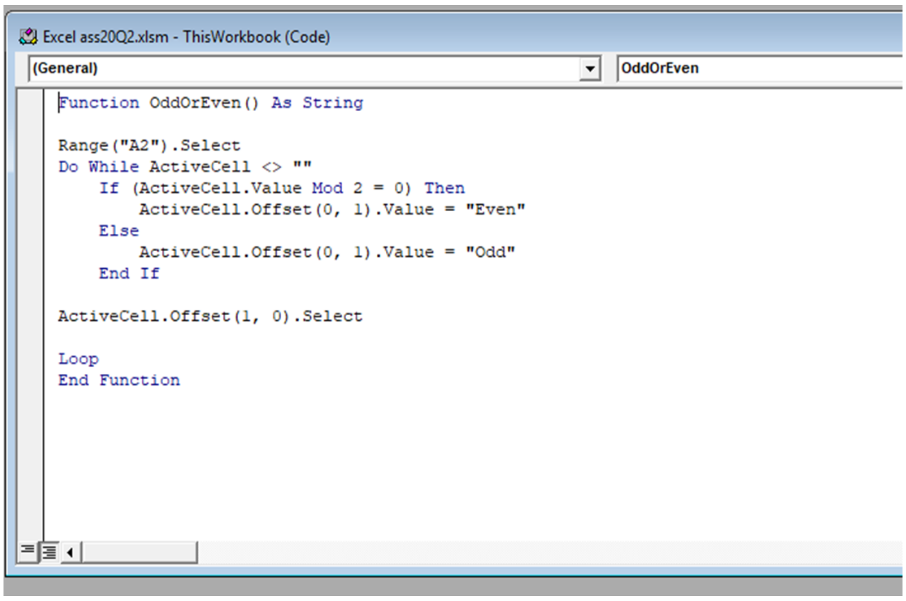


```
Sub Data_Analytics()

    Range("A5:C10").Select

End Sub
```

| Number | Odd or even |
|---|---|
| 56 | |
| 89 | |
| 26 | |
| 36 | |
| 75 | |
| 48 | |
| 92 | |
| 58 | |
| 13 | |
| 25 | |

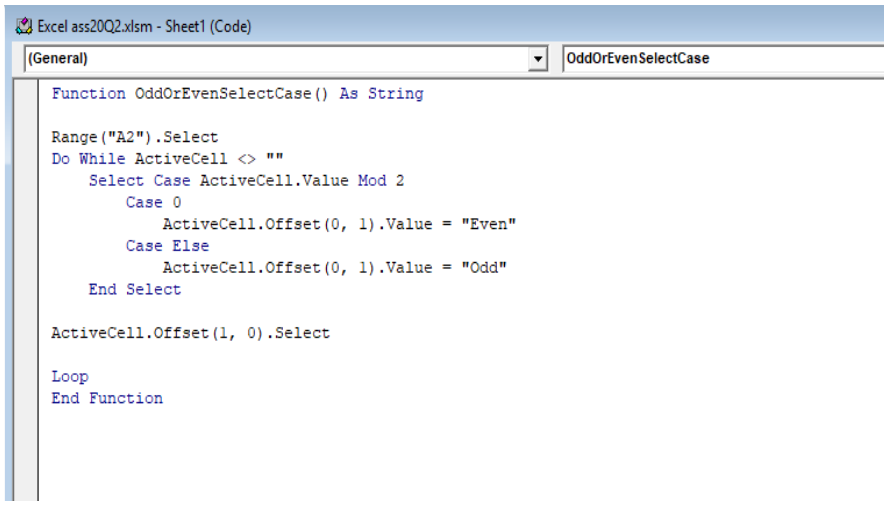
2. **Use the above data and write a VBA code using the following statements to display in the next column if the number is odd or even.**
   a. **IF ELSE statement**
   b. **Select Case statement**
   c. **For Next Statement**

a. IF ELSE statement



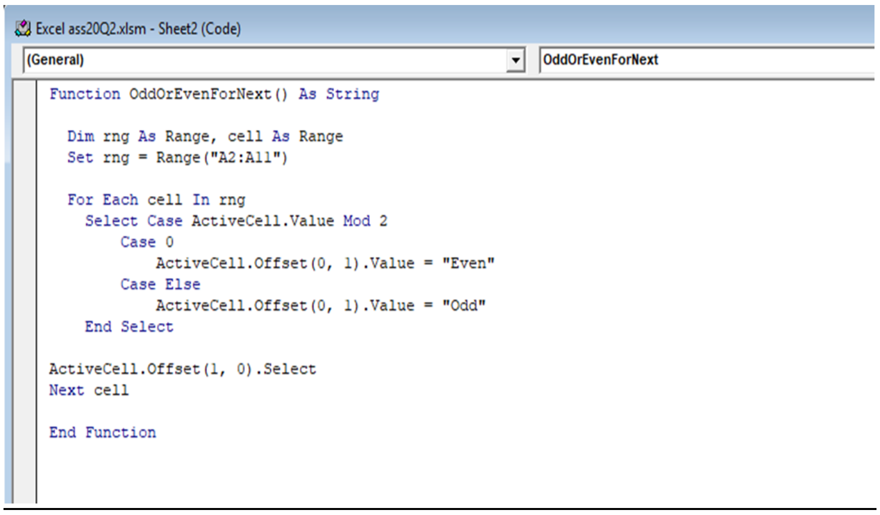
```
Function OddOrEven() As String

Range("A2").Select
Do While ActiveCell <> ""
    If (ActiveCell.Value Mod 2 = 0) Then
        ActiveCell.Offset(0, 1).Value = "Even"
    Else
        ActiveCell.Offset(0, 1).Value = "Odd"
    End If

ActiveCell.Offset(1, 0).Select

Loop
End Function
```

b. Select Case statement



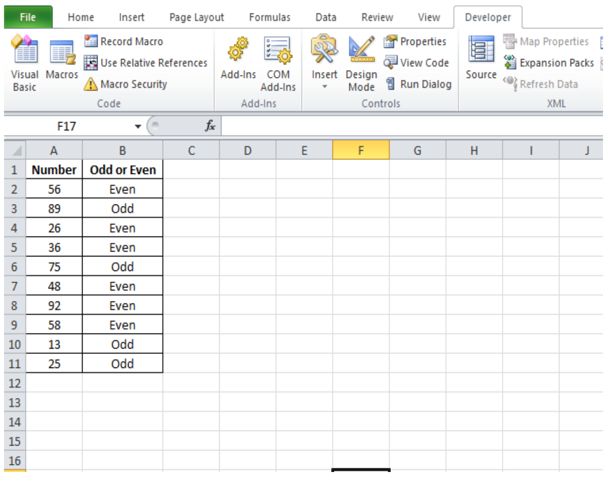
```
Function OddOrEvenSelectCase() As String

Range("A2").Select
Do While ActiveCell <> ""
    Select Case ActiveCell.Value Mod 2
        Case 0
            ActiveCell.Offset(0, 1).Value = "Even"
        Case Else
            ActiveCell.Offset(0, 1).Value = "Odd"
    End Select

ActiveCell.Offset(1, 0).Select

Loop
End Function
```

c. For Next Statement



```
Function OddOrEvenForNext() As String

  Dim rng As Range, cell As Range
  Set rng = Range("A2:A11")

  For Each cell In rng
    Select Case ActiveCell.Value Mod 2
        Case 0
            ActiveCell.Offset(0, 1).Value = "Even"
        Case Else
            ActiveCell.Offset(0, 1).Value = "Odd"
    End Select

ActiveCell.Offset(1, 0).Select
Next cell

End Function
```

Output



| Number | Odd or Even |
|---|---|
| 56 | Even |
| 89 | Odd |
| 26 | Even |
| 36 | Even |
| 75 | Odd |
| 48 | Even |
| 92 | Even |
| 58 | Even |
| 13 | Odd |
| 25 | Odd |

3. **What are the types of errors that you usually see in VBA?**

In Visual Basic, errors fall into one of three categories: syntax errors, run-time errors, and logic errors.

Syntax Errors

Syntax errors are those that appear while you write code. If you're using Visual Studio, Visual Basic checks your code as you type it in the Code Editor window and alerts you if you make a mistake, such as misspelling a word or using a language element improperly. If you compile from the command line, Visual Basic displays a compiler error with information about the syntax error. Syntax errors are the most common type of errors. You can fix them easily in the coding environment as soon as they occur.

Run-Time Errors

Run-time errors are those that appear only after you compile and run your code. These involve code that may appear to be correct in that it has no syntax errors, but that will not execute. For example, you might correctly write a line of code to open a file. But if the file does not exist, the application cannot open the file, and it throws an exception. You can fix most run-time errors by rewriting the faulty code or by using exception handling, and then recompiling and rerunning it.

Logic Errors

Logic errors are those that appear once the application is in use. They are most often faulty assumptions made by the developer, or unwanted or unexpected results in response to user actions. For example, a mistyped key might provide incorrect information to a method, or you may assume that a valid value is always supplied to a method when that is not the case. Although logic errors can be handled by using exception handling (for example, by testing whether an argument is Nothing and throwing an ArgumentNullException), most commonly they should be addressed by correcting the error in logic and recompiling the application.

4. **How do you handle Runtime errors in VBA?**

   Trapping Runtime Errors:

   The first step in dealing with runtime errors is to set a "trap" to catch the error. You do this by including an On Error statement in your macro. When a runtime error occurs, the On Error statement transfers control to an error-handling routine.

   To trap errors, you must set your error trap above the point in the procedure where errors are likely to occur. A good practice is to place the error trap near the top of the procedure. To avoid having the error-handling routine execute even when no error occurs, you should include an Exit Sub or Exit Function statement just before the error-handling routine's label.

```
Sub ErrorDemo ()
  On Error GoTo MyHandler
  'Program code goes here. To avoid invoking the error handling routine after this
  'section is executed, include the following line just above the error-handler.
  Exit Sub

  MyHandler:
  'Error is handled and the macro terminates gracefully.
  Exit Sub
End Sub
```

5. **Write some good practices to be followed by VBA users for handling errors**.

   Here are some best practices you can use when it comes to error handling in Excel VBA.

   Use ‘On Error Go [Label]’ at the beginning of the code. This will make sure any error that can happen from there is handled.

   Use ‘On Error Resume Next’ ONLY when you’re sure about the errors that can occur. Use it with expected error only. In case you use it with unexpected errors, it will simply ignore it and move forward. You can use ‘On Error Resume Next’ with ‘Err.Raise’ if you want to ignore a certain type of error and catch the rest.

When using error handlers, make sure you're using Exit Sub before the handlers. This will ensure that the error handler code is executed only when there is an error (else it will always be executed).

Use multiple error handlers to trap different kinds of errors. Having multiple error handler ensures that an error is properly addressed. For example, you would want to handle a 'type mismatch' error differently than a 'Division by 0' run-time error.

6. **What is UDF? Why are UDF's used? Create a UDF to multiply 2 numbers in VBA**

Microsoft offers us many built-in functions to speed up the work in Excel. However, we can create our functions using VBA coding, which is technically called "User-Defined Functions" (UDF). They are also called "custom functions" in Excel VBA.

Any formula we can access from the worksheet with a piece of code is called UDF. In simple terms, any formula not built-in but available in Excel is called "User Defined Functions."

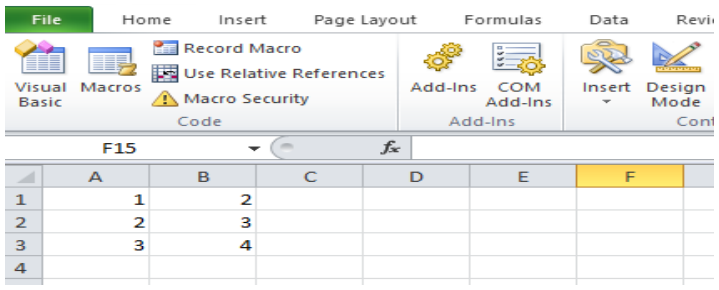
Even though UDF is part of our module, they are not part of our regular Subroutine in VBA

. It is called a Function procedure in VBA. Like how we start the macro coding with the word SUB similarly, we need to start this by using the word "Function." Sub procedure has a start and End. Similarly, the Function procedure has "Start" and "End."

UDF to multiply 2 numbers in VBA: In this code the function performs multiplication for every 2 numbers given in a row until it reaches the end of given data set.

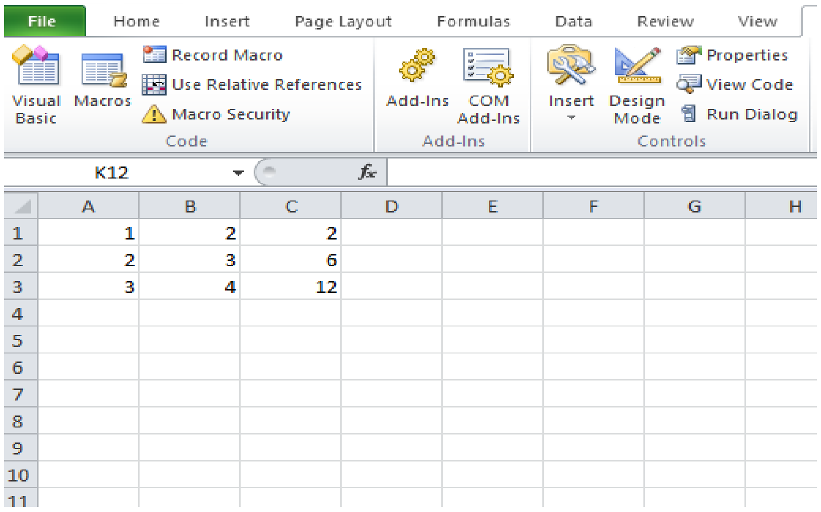
```
Function Mul() As Double
Dim C As Integer
Dim R As Integer
ActiveCell.Offset(0, 0).Range("A1").Select

Do While ActiveCell <> ""
        C = ActiveCell.Column
        R = ActiveCell.Row

        Mul = Cells(R, C) * Cells(R, C + 1)
        Cells(R, C + 2) = Mul
        ActiveCell.Offset(1, 0).Select
Loop
End Function
```

Input dataset:



|   | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | 1 | 2 |  |  |  |  |
| 2 | 2 | 3 |  |  |  |  |
| 3 | 3 | 4 |  |  |  |  |
| 4 |  |  |  |  |  |  |

Output of multiplication function:



|   | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 |  |  |  |  |  |
| 2 | 2 | 3 | 6 |  |  |  |  |  |
| 3 | 3 | 4 | 12 |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |