VBA 1: Foundations

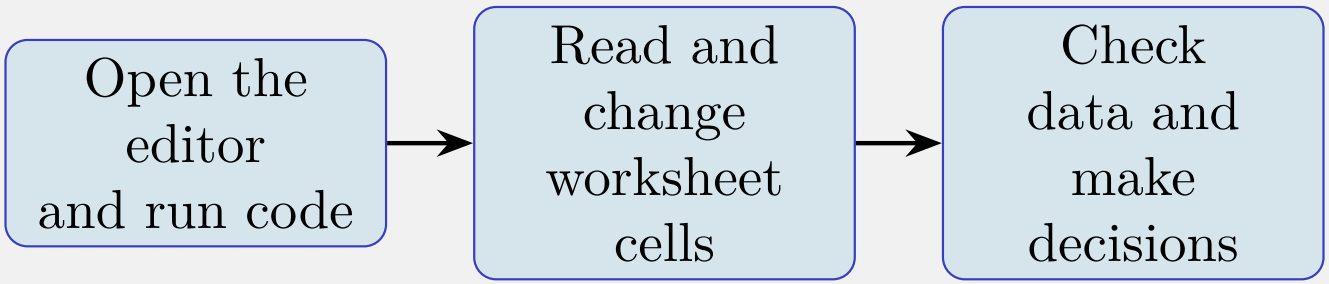
The editor, the language and the worksheet

Automated Decision Making in Business and Economics

From a worksheet to a small program

Visual Basic for Applications

VBA lets us write a list of instructions for Excel to follow. We will start with a greeting, then read cells, calculate a result and apply a simple rule.



Before starting: where does VBA run?

Environment	What to use in this course
Excel desktop: Windows	VBA editor and macros are available.
Excel desktop: Mac	VBA editor and macros are available; menus and keyboard shortcuts differ.
Excel for the web	It cannot create, edit or run VBA macros. Open the file in desktop Excel.

Three containers to distinguish

A **workbook** is the Excel file. Its **VBA project** holds code. A **module** is a container for related procedures inside that project.

Our examples use standard Excel objects, so the core code works on both Windows and Mac.

Windows: reveal the Developer tab

1. Open desktop Excel and a blank workbook.
2. Choose **File** → **Options** → **Customize Ribbon**.
3. Under **Main Tabs**, select **Developer**; click **OK**.
4. Choose **Developer** → **Visual Basic**.

Menu route in Windows Excel

A useful shortcut

Alt+F11 switches between Excel and its VBA editor.

Tab appearance depends on the version.

Mac: reveal the Developer tab

1. **Excel → Preferences → Ribbon & Toolbar.**
2. Under **Main Tabs**, tick **Developer.**
3. Click **Save.**
4. Select **Developer → Visual Basic.**

Alternative editor route:
Tools → Macro → Visual Basic Editor.

Labels and appearance may vary by release.

Select Developer

Excel for Mac: Ribbon & Toolbar

Save the workbook with its code

Extension	Meaning	Stores VBA?
`.xlsx`	Excel workbook	No
`.xlsm`	Excel macro-enabled workbook	Yes
`.bas`	Exported VBA module text	Code only

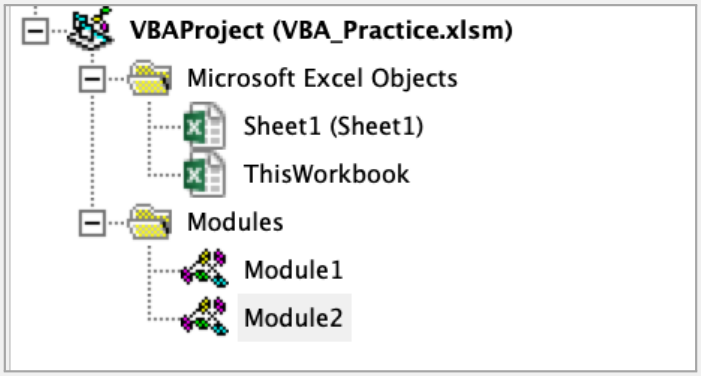
1. Use **File** → **Save As**; choose **Excel Macro-Enabled Workbook (.xlsm)**.
2. Save as `VBA_Practice.xlsm`.
3. When reopening, enable macros only if you know and trust their source (for example: you!).

Why the file type matters

Saving the final file as `.xlsx` removes its VBA project!

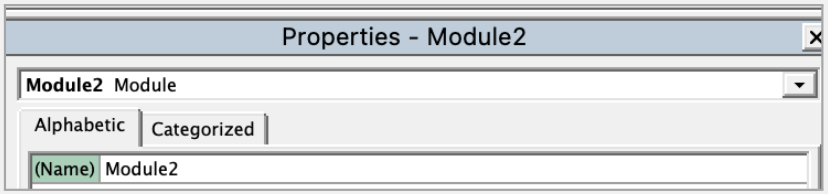
The editor: three areas, three jobs

1. Project Explorer



Choose the workbook, sheet or module.

2. Properties



See details of the selected item.

3. Code window

Write instructions in the selected module.

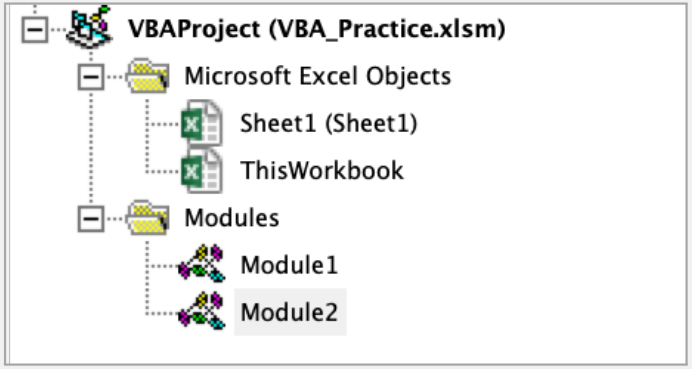
Missing Project Explorer? Choose **View** → **Project Explorer**.

Details from the Visual Basic Editor for Mac

Create a standard module

1. Select **VBAProject (VBA_Practice.xlsm)** in Project Explorer.
2. Choose **Insert** → **Module** in the editor.
3. A module appears under **Modules**; its number can vary.
4. Double-click it to type in its code window.

Inside the workbook



This example uses **Module2**.

Visual Basic Editor for Mac

Why a standard module?

Our first procedures are ordinary commands that we choose to run. Sheet modules and `ThisWorkbook` also support event code, which responds to events such as opening a file.

The code window after inserting a module

Visual Basic Editor for Mac: the code window in Module2

- The selected module belongs to the workbook's **VBA project**.
- The **code window** contains one named procedure, `SayHello`.
- Statements go between `Sub SayHello()` and `End Sub`. The next slide explains how to type and run them.

Your first procedure: say hello

Type the following into the standard module:

```
Option Explicit

Sub SayHello()
    MsgBox "Hello, Automated Decision Making!"
End Sub
```

1. Put the cursor anywhere inside `SayHello`.
2. Choose **Run** → **Run Sub/UserForm** in the editor.
3. Read the message; click **OK** to close it.

What just happened?

Excel executed the statements in the procedure. The message box waits for your response; then execution continues to `End Sub`.

Read the first program as a sentence

Code	Meaning and reason for the name
`Sub`	Short for **subroutine**: a named procedure that performs actions.
`SayHello`	A name chosen by us; use a verb describing the action.
`()`	The parameter list; empty here because no inputs are supplied.
`MsgBox`	Abbreviation of **message box**.
`"Hello..."`	A string literal: the quoted text itself.
`End Sub`	Marks where this subroutine ends.

Indentation makes the structure visible. VBA is not case-sensitive: the editor normally standardizes keyword capitalization.

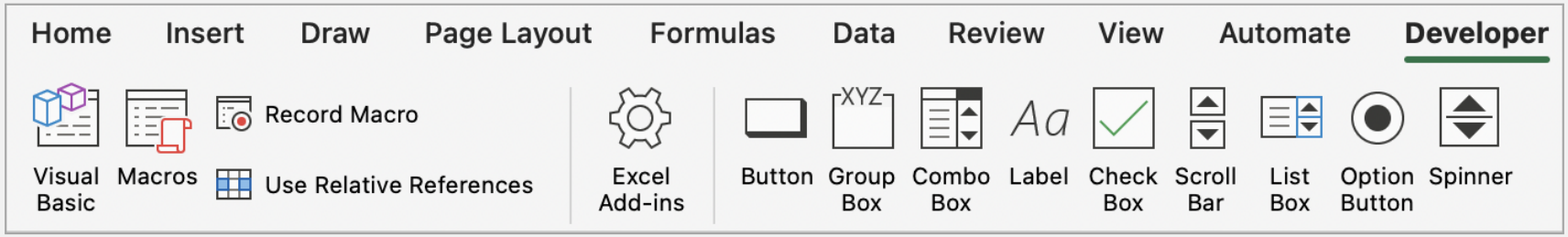
Run, pause and stop deliberately

Action	Reliable menu route
Run the current procedure	Editor: **Run** → **Run Sub/UserForm**.
Run from the workbook	Excel: **Developer** → **Macros**; select `SayHello`; choose **Run**.
Stop a paused procedure	Editor: **Run** → **Reset**, or the square Stop/Reset button.

An important consequence

Stopping code does not undo worksheet changes already made!
Before operating...always save a copy of important data.

Give the user a button



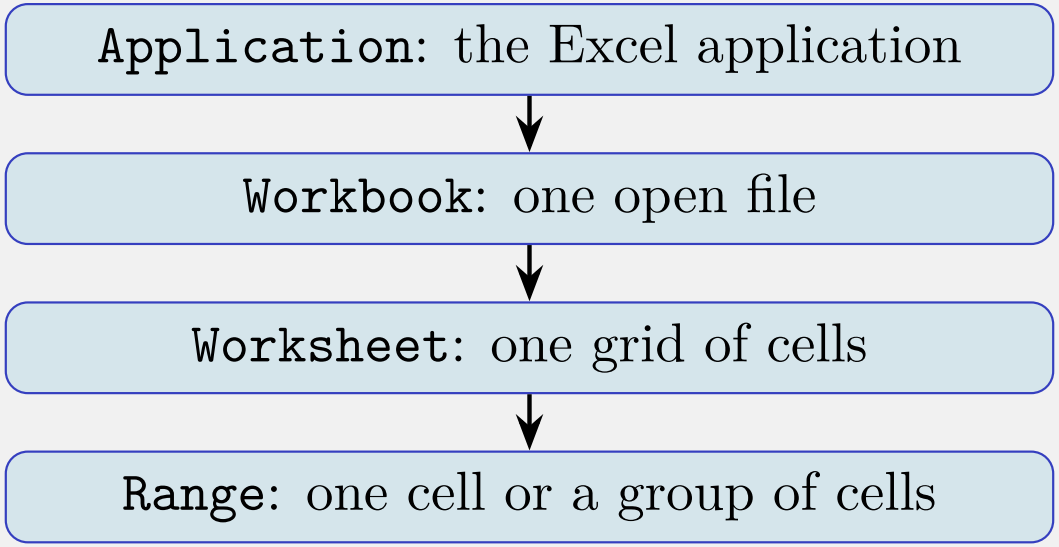
Excel for Mac: the Button control on the Developer tab

1. Return to the **worksheet**, not the code editor.
2. Windows: **Developer** → **Insert** → **Button (Form Control)**. Mac: choose the **Button** control on the Developer tab.
3. Drag on the sheet to draw the button.
4. In **Assign Macro**, choose `SayHello`; click **OK**.
5. Click the button to execute the same procedure.

Separate the interface from the rule

A button is an entry point. The instructions remain in the procedure. Use a Form Control button for this cross-platform example.

Excel is organised into files, sheets and cells



Why this hierarchy matters

"Write to B3" is incomplete if several workbooks are open. We identify the workbook, then the worksheet, then the range.

A **collection**, such as `Worksheets`, groups objects of the same general kind. `Worksheets("Practice")` selects one by name.

The dot means: inside this object

```
ThisWorkbook.Worksheets("Practice").Range("B3").Value2
```

Part	Interpretation
`ThisWorkbook`	The workbook containing the running code.
`.Worksheets("Practice")`	Its worksheet named Practice.
`.Range("B3")`	Cell B3 on that worksheet.
`.Value2`	The value stored in the cell.

Properties and methods

A **property** is an attribute, such as `Value2` or `Name`. A **method** is an action: `ws.Range("B7").ClearContents` clears the contents of B7 while keeping its formatting.

Which workbook are we changing?

`ThisWorkbook`

The file that contains this code.

If our module is in `VBA_Practice.xlsm`, this reference keeps pointing to that file even when another file becomes active.

`ActiveWorkbook`

The workbook currently active in Excel.

It may change when the user clicks another window or when code opens another file.

Course convention: put the module and the practice data in the same workbook; use `ThisWorkbook`.

A frequent hidden error

Unqualified `Range("B3")` in a standard module refers to the active sheet. Correct-looking code can therefore change the wrong file.

Give the worksheet a short name with Set

```
Dim ws As Worksheet
Set ws = ThisWorkbook.Worksheets("Practice")
ws.Range("B3").Value2 = 12
```

- `Dim ws As Worksheet` declares an object variable.
- `Set` assigns an **object reference**; it does not copy the sheet.
- Assigning an ordinary value uses `=` without `Set`.

The sheet must exist, and its name must match.

Range and Cells: addresses and coordinates

```
ws.Range("B3").Value2 = 12
ws.Cells(3, 2).Value2 = 12

ws.Range("B3:B5").Font.Bold = True
```

`Cells(row, column)` uses row first. Column 2 is B. Both first lines refer to the same cell.

	A	B
1		
2		
3	Units	12
4	Price	8.50
5	Cost	5.00

Which notation is useful?

`Range("B3")` is easy to read when the address is fixed. `Cells(r, 2)` is useful when a loop changes the row number `r`.

A value and a formula are different things

```
Dim units As Double
units = ws.Range("B3").Value2

ws.Range("B7").Value2 = 42
ws.Range("B7").Formula = "=B3*(B4-B5)"
```

Operation	Consequence
Read `.Value2`	Obtain the cell's value or a formula's calculated result.
Write `.Value2`	Store a value; replace any existing formula.
Write `.Formula`	Store an Excel formula.

`Value2` is a property name, not "value squared". Unlike `Value`, it does not use VBA's Currency and Date subtypes; numeric dates are read as serial numbers.

Variables: named places for intermediate values

Variables keep the values we need while the macro is running; their names tell us what those values mean.

```
Option Explicit

Sub VariableDemo()
    Dim units As Double
    Dim price As Double
    units = 12
    price = 8.5
    MsgBox units * price
End Sub
```

- `Dim` declares a variable; its historical name comes from *dimension*. Here it means "introduce this variable".
- `As Double` states the type of value it should hold.
- `Option Explicit`, at the top of a module, requires declarations: a typo such as `untis` becomes a detectable error.

Choose a type that matches the meaning

Type	Intended use	Example
`Long`	Whole-number counters and row indices	Row 100000
`Double`	Floating-point numerical calculations	Price 8.50
`String`	Text	`"Review"`
`Boolean`	A logical value	`True` or `False`
`Date`	A date and time	`DateSerial(2026,9,19)`
`Variant`	A value whose type may vary	Raw worksheet input

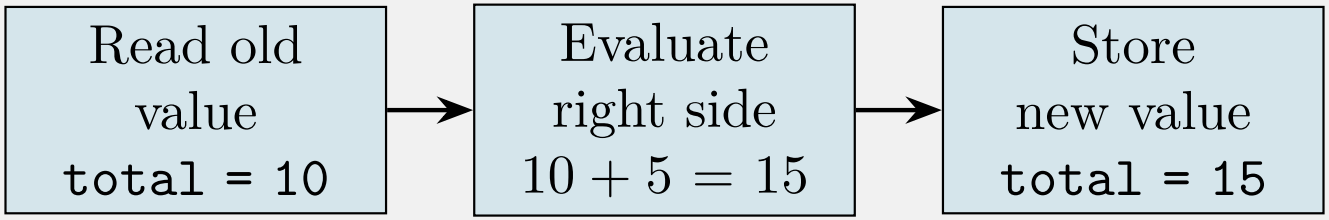
`DateSerial(year, month, day)` constructs a date without an ambiguous text format.

Two practical choices

Use `Long`, not `Integer`, for Excel row indices: VBA Integer stops at 32767. Use `Variant` when inspecting a cell that might contain a number, text or an Excel error.

Assignment is an update, not an equation

```
Dim total As Double
total = 10
total = total + 5
```



The equation symbol in two contexts

In `total = total + 5`, the symbol `=` is assigning a value.
In `If total = 15 Then`, the symbol `=` is comparing two values and returns a Boolean result.

An apostrophe starts a comment: `' Add the new amount.`

Operators and constants express the rule

```
Const TargetProfit As Double = 40
profit = units * (price - cost)
message = "Profit: " & CStr(profit)
```

Symbol / word	Meaning
`+ - * / ^`	Add, subtract, multiply, divide, raise to a power.
`= <> < <= > >=`	Equal, unequal and ordered comparisons.
`And Or Not`	Combine or negate Boolean conditions.
`&`	Concatenate: join pieces of text.

`Const` declares a constant: the code cannot assign it another value.

`CStr` means convert to String.

Our example: from sales data to a decision

We use a small worksheet for one sales scenario. We know the **quantity sold**, the **selling price per unit** and the **cost per unit**.

What do we want to find out?

Calculate the profit and check whether it reaches the **target of 40**.

$$\text{Profit} = \text{Units} \times (\text{Price} - \text{Unit cost}).$$

`B3:B5` hold the inputs. The macro will write profit in `B7` and the decision in `B8`. With our numbers, profit is **42**, so the target is met.

Build the Practice worksheet

Create a worksheet named **Practice** and enter the following inputs.

	A	B	Meaning
3	Units	12	Quantity sold
4	Price	8.50	Revenue per unit
5	Unit cost	5.00	Cost per unit
7	Profit		Output written by code
8	Decision		Output written by code

Where the inputs and results go

B3:B5 contain nonnegative numeric values; fractional quantities are allowed. B7:B8 are designated outputs. The macro changes these output cells, so they must be empty.

Read, calculate, write: the first useful macro

```
Sub CalculateProfit()
    Dim ws As Worksheet
    Dim units As Double, price As Double, cost As Double
    Dim profit As Double
    Set ws = ThisWorkbook.Worksheets("Practice")
    units = ws.Range("B3").Value2
    price = ws.Range("B4").Value2
    cost = ws.Range("B5").Value2
    profit = units * (price - cost)
    ws.Range("B7").Value2 = profit
End Sub
```

Follow the data

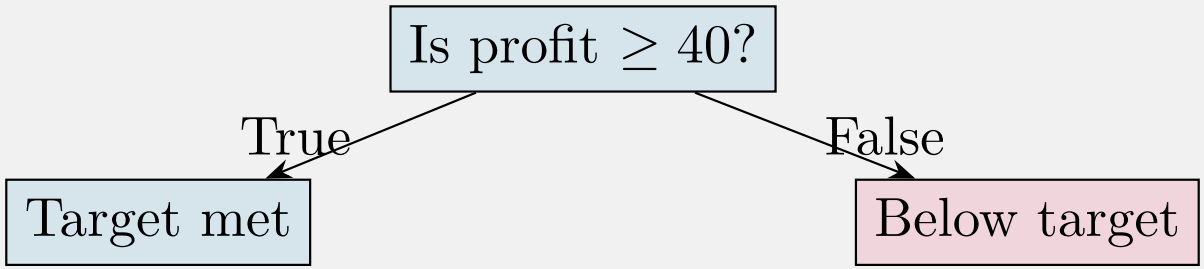
Worksheet → variables → calculation → worksheet. With the supplied inputs, B7 becomes 42.

This first version assumes the inputs are valid. We will add checks before trusting data supplied by someone else.

If ... Then ... Else: choose between two actions

Suppose a case meets the target if its profit is at least 40.

```
If profit >= 40 Then
    ws.Range("B8").Value2 = "Target met"
Else
    ws.Range("B8").Value2 = "Below target"
End If
```



Only one branch runs. `ElseIf` can add another condition before `Else`; `End If` closes the whole decision.

Check inputs in a safe order

```
Dim v As Variant
v = ws.Range("B3").Value2
If IsError(v) Then
    MsgBox "B3 contains an Excel error."
    Exit Sub
End If
If Not IsNumeric(v) Then
    MsgBox "B3 must contain a number."
    Exit Sub
End If
If CDbl(v) < 0 Then
    MsgBox "B3 cannot be negative."
    Exit Sub
End If
```

`IsError` and `IsNumeric` ask questions; `CDbl` converts to Double; `Exit Sub` ends the procedure early.

For ... Next: repeat a known number of times

We now read B3, B4 and B5 in turn, instead of writing the same instruction three times.

```
Dim r As Long
For r = 3 To 5
    Debug.Print r, ws.Cells(r, 2).Value2
Next r
```

- `For` introduces a counter-controlled loop.
- `r` starts at 3 and increases by 1 after each pass.
- The body runs for `r = 3`, then 4, then 5. Both limits are included.
- `Next r` means: update the counter and test whether another pass is needed.

Observe without changing cells

`Debug.Print` sends information to the editor's Immediate window: choose **View** → **Immediate Window**. It does not print a worksheet or a paper page.

Trace a loop before you run it

```
Dim k As Long, total As Double
total = 0
For k = 1 To 3
    total = total + k
Next k
```

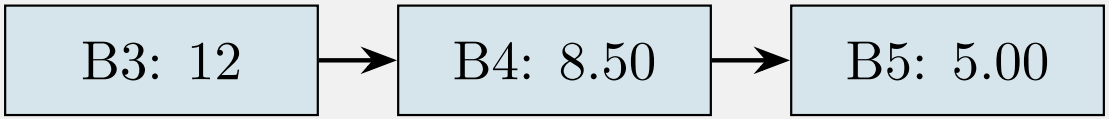
Pass	`k`	`total` before	`total` after
1	1	0	$0 + 1 = 1$
2	2	1	$1 + 2 = 3$
3	3	3	$3 + 3 = 6$

Keep a running total

Initialize once, **before** the loop. Update inside it. Setting `total = 0` inside the loop would discard the previous passes.

For Each: work through the cells one by one

```
Dim cell As Range
For Each cell In ws.Range("B3:B5")
    cell.Interior.Color = RGB(230, 240, 255)
Next cell
```



`cell` refers to one cell at a time. No numerical counter is necessary. `Interior` describes the cell's background; `RGB` specifies red, green and blue intensities from 0 to 255.

Choose the loop that reflects the problem

Use `For` when the row number matters. Use `For Each` when the task is naturally "do this to every cell or sheet".

Do While: repeat while a condition is true

```
Dim amount As Double, steps As Long
amount = 10
steps = 0
Do While amount < 20 And steps < 100
    amount = amount * 1.1
    steps = steps + 1
Loop
Debug.Print amount, steps
```

- The condition is checked **before** each pass; zero passes are possible.
- Multiplying by 1.1 represents an increase of 10% per step.
- The result first exceeds 20 after 8 steps: approximately 21.44.

Sub and Function answer different questions

	`Sub`	`Function`
Main purpose	Perform an action	Return a value
Example name	`EvaluatePractice`	`UnitMargin`
Typical job	Read data; write results	Compute price minus cost
Called using	Its name and arguments	Its name inside an expression
Ending	`End Sub`	`End Function`

Separate calculation from interaction

A function that only calculates is easy to check with known inputs. A procedure can manage the worksheet and call that function when it needs the result.

A reusable function: unit margin

```
Public Function UnitMargin(ByVal price As Double, _
                           ByVal cost As Double) As Double
    UnitMargin = price - cost
End Function
```

- `Public` allows calls from elsewhere in the project and, here, from worksheet formulas.
- `price` and `cost` are parameters: named inputs.
- The final `As Double` describes the returned value.
- Assign to the function's own name to provide that return value.

How a call works

`UnitMargin(8.5, 5)` returns 3.5. Thus
`profit = 12 * UnitMargin(8.5, 5)` assigns 42 to profit.

Space plus underscore, `_`, continues a long statement on the next line.

Use your function in a worksheet formula

Put `UnitMargin` in a **standard module** in the saved `.xlsm` workbook. In an empty output cell, enter:

```
=UnitMargin(B4,B5)
```

result: 3.50

Why this is called a UDF

A **user-defined function** adds a calculation whose definition you provide. Excel supplies the cell values as arguments and displays the returned value.

- Macros must be allowed for VBA UDFs to run.
- Some regional settings use semicolons between formula arguments.
- A function called by a worksheet formula should calculate and return a result; it must not attempt to edit other cells or format the workbook.
- Pass required inputs as arguments so Excel can track their dependencies.

Two different kinds of error

Kind	Example	How to investigate
Syntax / compile error	Missing `End If`; misspelled undeclared variable	Read the highlighted line; use **Debug → Compile VBAProject**.
Runtime error	Requesting a sheet named Practice when it does not exist	Read the error message; inspect the object and data.

Find a mistake: run one line at a time

1. Click the left margin beside an executable statement: a **breakpoint** pauses execution before that line.
2. Run the procedure and wait for it to pause.
3. Use **Debug** $\rightarrow$ **Step Into** (usually F8) to execute one statement at a time.
4. Continue with Run/F5; use Reset to stop the paused procedure.