How can I get the sheet name using VBA?

Authored by

stats writer

November 18, 2025

Introduction to VBA and Dynamic Workbook Navigation

The ability to dynamically identify and reference specific worksheets is fundamental to writing robust and efficient VBA code. Whether you are building automated reports, creating custom user interfaces, or managing complex data migrations, knowing the name of the sheet you are interacting with is often the starting point. This guide details the essential methods within Visual Basic for Applications (VBA) that allow developers to accurately retrieve the display name of any worksheet within an Excel workbook. We will focus on two primary approaches: accessing the currently selected sheet and referencing a sheet based on its positional index.

Understanding the underlying structure of the Excel Object Model is paramount to mastering these techniques. Every element in an Excel file, from the application itself down to individual cells, is represented as an object with specific properties and methods. The sheet name, which is the text displayed on the tab at the bottom of the window, is stored as the Name property of the respective Worksheet object. Utilizing these properties through VBA functions ensures that your scripts can adapt dynamically to changes in the workbook structure, vastly improving the maintainability of your macros.

We will analyze two concise yet powerful Function procedures designed to handle the most common scenarios. These procedures are easily implemented into any standard module within the VBA Editor and can be called directly from worksheet cells as User Defined Functions (UDFs). By converting these routines into UDFs, end-users gain the power of VBA logic without needing to access the macro environment, enabling highly customized formulas that extend beyond standard Excel capabilities.

Understanding the Excel Object Model for Sheet Access

Before diving into the code, it is essential to grasp how VBA views the structure of an Excel file. The hierarchy is simple: the application contains workbooks, and each workbook contains a collection of worksheets. The key collection we interact with is the Sheets collection. When you want to retrieve information about a sheet, you must first reference the collection it belongs to, followed by the specific sheet object, and finally the property you wish to extract.

The core syntax for accessing a sheet's attributes typically involves referencing the collection and then using either the sheet's name string or its index number. For instance, to get the name of the third sheet in the current workbook, the code would be `ActiveWorkbook.Sheets(3).Name`. This structure ensures unambiguous communication between your VBA code and the Excel environment. The property we are primarily interested in for this task is the **.Name** property, which returns the string displayed on the sheet tab.

The methods discussed below provide two distinct ways to target the sheet object. The first

method leverages the `ActiveSheet` property, which is implicitly available and useful when the code's purpose is to act upon the user's current focus. The second method uses the `Sheets()` collection indexer, offering a powerful way to reference sheets based on their position, regardless of which sheet is currently visible or active. Both methods rely on the robust object-oriented nature of VBA to perform precise data retrieval.

Strategy 1: Retrieving the Name of the ActiveSheet

The simplest and most direct way to get a sheet's name is by using the ActiveSheet property. This property refers specifically to the worksheet that is currently visible and selected by the user. When a user clicks on a sheet tab, that sheet becomes the active sheet, and any operation referencing `ActiveSheet` will be directed to it. This approach is highly effective in scenarios where the code is triggered by a user action on a specific sheet, such as a button click or a worksheet event.

To implement this, we define a Function that simply accesses the ActiveSheet object and retrieves its **Name** property. Since the ActiveSheet property always returns a single worksheet object, the code is extremely clean and concise. The resulting value is a standard text string (String data type) corresponding exactly to the label on the sheet tab. This method avoids the need for explicit sheet referencing, making the code portable across different workbooks where sheet names might vary.

The following procedure, defined as a User Defined Function (UDF), demonstrates this mechanism. When called from a cell in Excel, the function executes, identifies the sheet containing the formula, and returns that sheet's name. This simple utility allows for self-referencing formulas, which can be invaluable when creating headers or dynamic titles that must reflect the current context of the sheet.

Implementing the ActiveSheet Method: Code Breakdown and Usage

The implementation for retrieving the active sheet's name is straightforward and requires minimal lines of code. We define a public Function named `GetSheetName`. Within this function, we assign the return value (which will be the name of the sheet) directly from the `ActiveSheet.Name` property. This code is placed within a standard VBA Module.

Function GetSheetName()

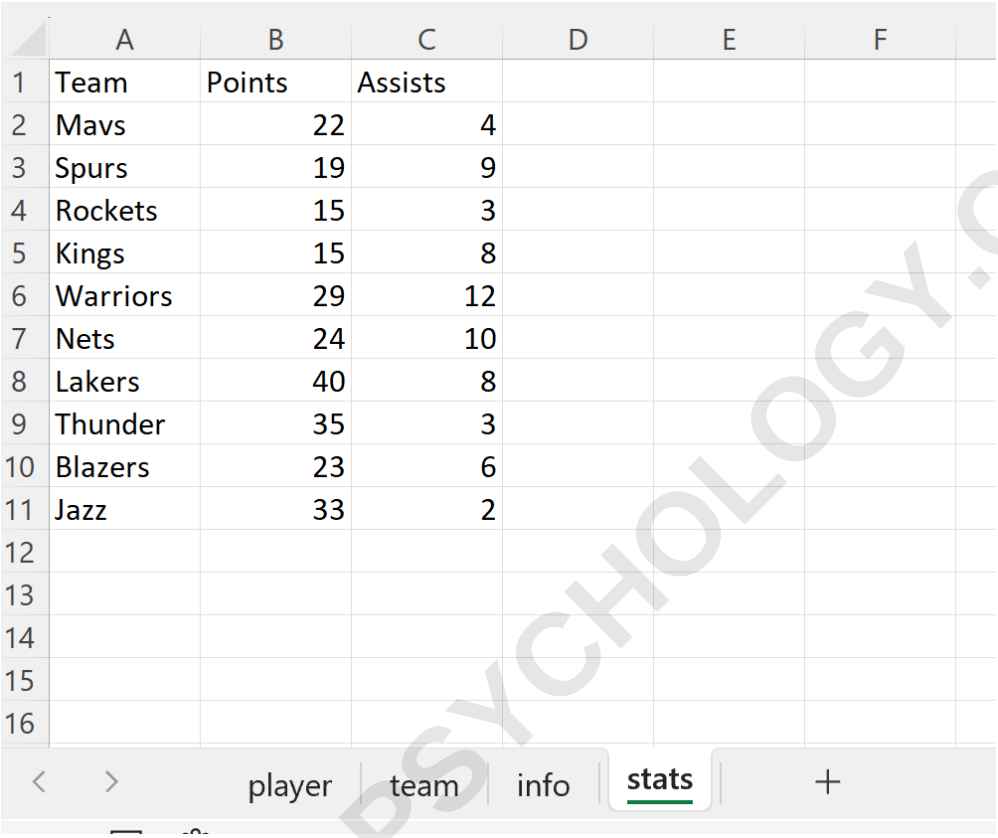
GetSheetName = ActiveSheet.Name

End Function

This code snippet defines the procedure for retrieving the name of the active sheet. When this UDF is called from any cell in the workbook, it executes the single assignment line, effectively returning

the name of the sheet where the formula resides, provided that sheet is currently active. If the workbook is open and has four sheets named 'Data', 'team', 'stats', and 'Reports', as illustrated below, the result will depend entirely on which sheet the user has selected.

The following examples illustrate the workbook structure we are using for demonstration purposes. The workbook contains four distinct sheets, each with its own unique name.



	A	B	C	D	E	F
1	Team	Points	Assists			
2	Mavs	22	4			
3	Spurs	19	9			
4	Rockets	15	3			
5	Kings	15	8			
6	Warriors	29	12			
7	Nets	24	10			
8	Lakers	40	8			
9	Thunder	35	3			
10	Blazers	23	6			
11	Jazz	33	2			
12						
13						
14						
15						
16						

Suppose, for instance, that the sheet labeled **stats** is the currently active sheet in the workbook. If a user types the formula `=GetSheetName()` into cell **E1** of the **stats** sheet, the function will evaluate the ActiveSheet property at the time of calculation and return the string value associated with its name.

Example 1: Use VBA to Get Name of ActiveSheet

Let us utilize the function defined above to concretely demonstrate its application. We assume the function `GetSheetName()` has been properly added to a module. When we call this formula within the currently active sheet, the result is instantaneous and accurate.

Function GetSheetName()

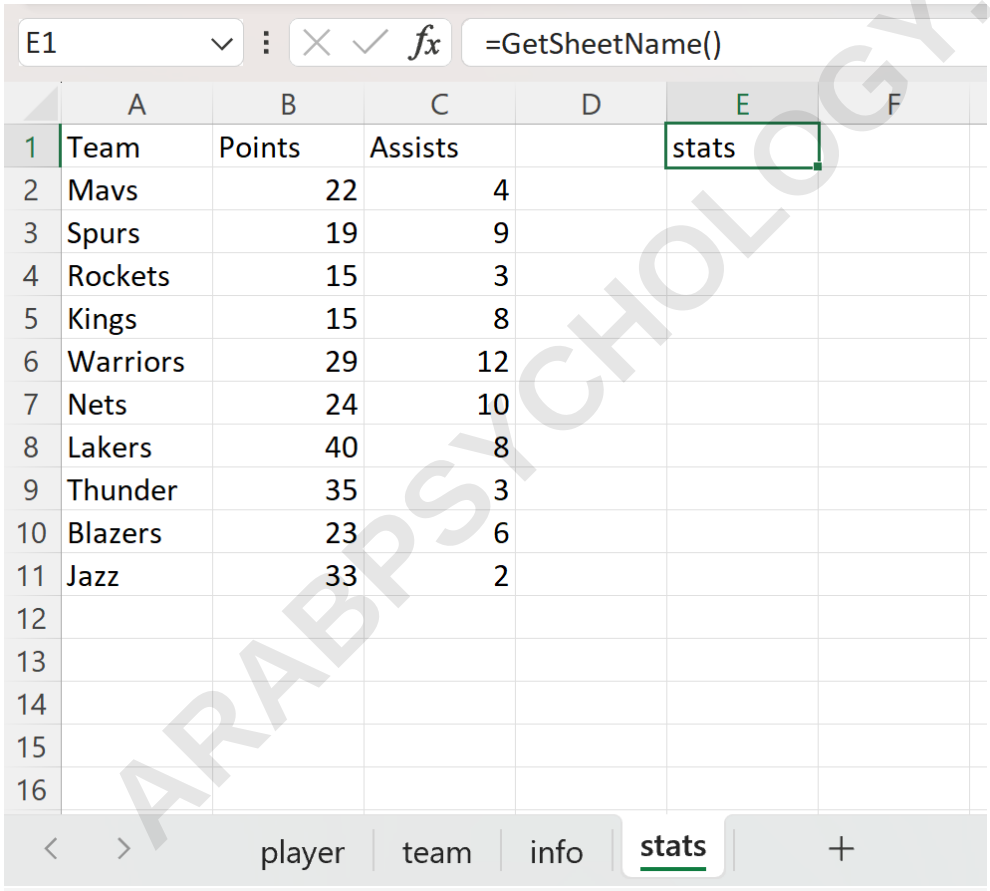
GetSheetName = ActiveSheet.Name

End Function

As specified, if the sheet named **stats** is currently active, we can invoke the UDF directly using the following formula typed into cell **E1** of that sheet:

=GetSheetName()

Upon calculation, the function resolves the name of the active worksheet and displays it in cell E1. This is particularly useful for generating dynamic report headers or for use in conditional formatting rules that depend on the sheet's identity. The following screenshot visually confirms the execution of this formula:



The output confirms that since **stats** was the active sheet when the formula was executed, the function successfully returned that specific string. This method is the easiest to implement but is limited to the sheet currently in focus. For scenarios requiring reference to non-active sheets, we must employ the second strategy: referencing by index.

Strategy 2: Identifying Sheets by Index Number

While using the ActiveSheet property is convenient, sometimes it is necessary to retrieve the name of a specific sheet regardless of whether it is currently active. For this purpose, VBA provides the ability to reference sheets using their index number. The index number corresponds to the sheet's physical position in the workbook, counting from left to right (e.g., the leftmost tab is index 1, the second is index 2, and so on).

To retrieve the name using the index, we utilize the Sheets collection and pass the desired positional number as an argument. The syntax is `Sheets(N).Name`, where `N` is the Integer index. This method offers stability in environments where sheet names are prone to change, provided that the physical order of the sheets remains constant. It is crucial to remember that Excel indexing is 1-based, meaning the counting starts at 1, not 0.

This approach requires the UDF to accept an input parameter, which dictates which sheet index to target. By formalizing this input as an Integer, we ensure that the function is robust and only accepts valid numeric positions. This makes the UDF highly reusable across various workbooks and allows users to quickly reference any sheet by its position in the tab order, even if they do not know the sheet's current name.

Practical Application: Using Index Numbers for Dynamic Sheet Referencing

To implement sheet referencing by number, we modify our previous Function to accept a single Integer argument, which represents the index we want to target. This index is then passed directly to the `Sheets()` collection property.

Function GetSheetName(N As Integer)

GetSheetName = Sheets(N).Name

End Function

In this new structure, `N As Integer` designates that the input parameter `N` must be a whole number, signifying the position of the desired sheet. The function then returns the `.Name` property of the sheet object located at that specific index. This is an incredibly flexible tool for generating summaries or dashboards that pull information from multiple, ordered source sheets.

Example 2: Use VBA to Get Name of Sheet by Number

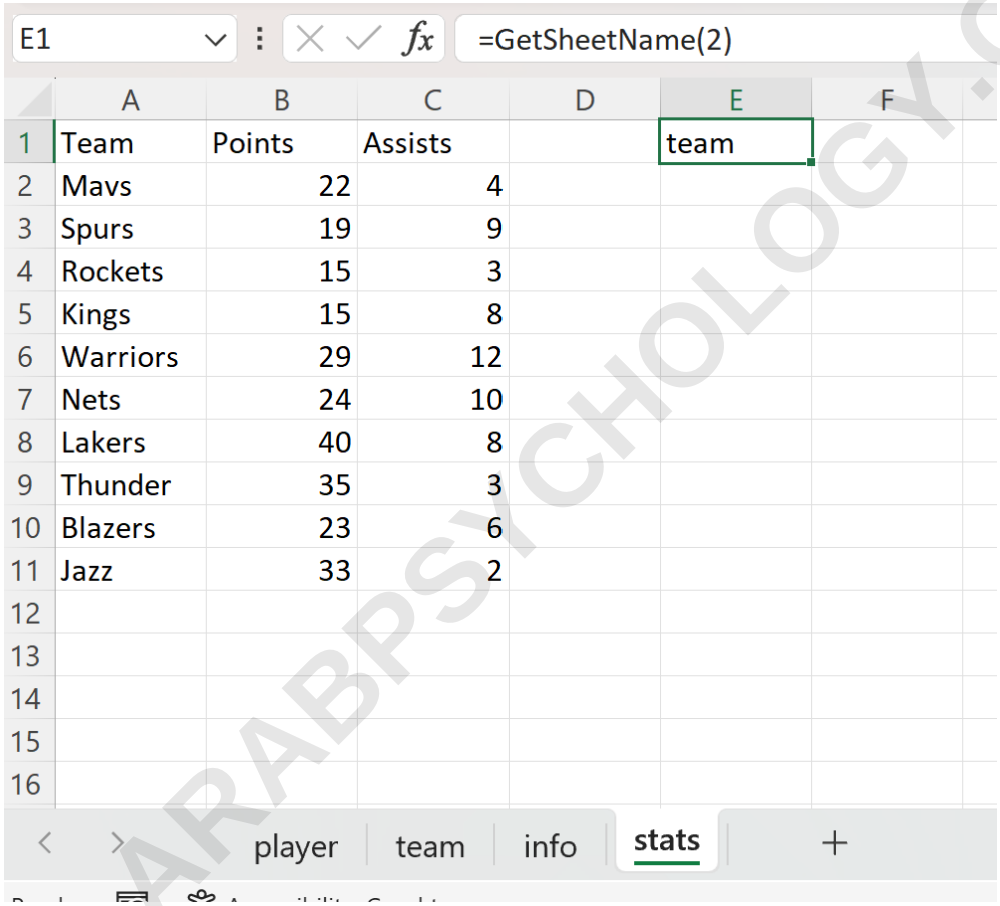
Following the creation of the index-based function, we can now test its utility by targeting a sheet based on its position. Referring back to our example workbook, which has sheets ordered as 'Data'

(1), 'team' (2), 'stats' (3), and 'Reports' (4), let us retrieve the name of the second sheet.

Once the function is created, we can type the following formula into cell **E1** of the currently active sheet (it does not matter which sheet is active, as we are referencing by absolute position):

=GetSheetName(2)

This call instructs the VBA function to look into the workbook's sheet collection and extract the name of the sheet located at position 2. The formula's output is independent of the sheet where the formula resides. The following screenshot confirms the result of this operation:



The function successfully returns the value **team**, confirming that this is the name of the second sheet in the workbook's current tab order. This demonstrates the power of index-based referencing for retrieving information from non-active sheets.

Advanced Considerations: Looping Through All Sheet Names

While referencing sheets individually is helpful, most advanced VBA applications require iterating through the entire collection of sheets. This is achieved using a loop structure, typically a `For Each`

loop, which is often the most efficient way to process multiple items in a collection. By looping through the Sheets collection, we can extract and process the names of all worksheets sequentially.

A common application for this technique is generating a table of contents or performing a consistent data validation check across all available sheets. The code below demonstrates how to iterate through all sheets in the active workbook and print their names to the Immediate Window (accessible via Ctrl+G in the VBA Editor).

```
Sub ListAllSheetNames()

Dim ws As Worksheet

For Each ws In ActiveWorkbook.Sheets
Debug.Print ws.Name
Next ws

End Sub
```

This procedure assigns each worksheet object in the collection to the variable `ws` sequentially. Inside the loop, `ws.Name` retrieves the name property for the current sheet, which is then outputted for review. This looping strategy is significantly more scalable than trying to reference dozens of sheets individually by index or name, forming the backbone of automated reporting systems within Excel.