Utility statements

UFT uses VBScript as its programming language, but it also has a wic array of commands that are very useful for various purposes, such flow control. In this section, we shall provide a list of commonly used commands and some others, which the authors think are useful, accompanied by examples and explanations.

Getting ready

From the **File** menu, navigate to **New** | **Test**, or use the *Ctrl + N* shortcut.

How to do it…

Proceed with the following steps:

1. `DescribeResult`: This returns a text description of the specifie error code. For example, to print the description of the last runti error, use:

```
Print DescribeResult(GetLastError())
```

2. `ExecuteFile`: This is used to execute the VBScript code in a file (function library) during runtime. This means, instead of associating a function library to your test during design time via the UFT GUI (via **File** | **Settings** | **Resources**), you can call it directly using the following syntax:

```
ExecuteFile FilePathname
```

3. After the statement is executed, all the definitions (variables, constants, functions, subroutines, and classes) in the file are available from the global scope of the action's script. It is, however, one of the very few statements that are highly recommended as not to be used. The risk is obvious; loading code in such a fashion might cause runtime errors (in the best case) or data overriding (in the worst case). For example, if an identifier (variable, constant, function, and so on) has already been loaded, then an error would arise, as duplicate definitions are not allowed in the same namespace. A worse scenario is if a global variable is reinitialized (assigned with a new value), thus exposing the test to unreliable results.
4. `ExitAction`: This is used to abort the execution of the current action. If the action is set to run for multiple iterations, the consequent iterations will not be executed. Control returns to the calling `Action` (for nested Actions) or to the next `Action`. If `Action` is the last to be run, then the test will stop.
5. `ExitActionIteration`: This is used to abort the execution of the current iteration of an action. When an `Action` is set to run for multiple iterations, control returns to the next iteration (the beginning of the `Action` script). If the iteration is the last, control returns to the calling `Action` (for nested actions) or to the next `Action`. If the `Action` is the last to be run, then the test will stop.
6. `ExitTest`: This is used to abort the execution of the current test. If the test (or the current `Action`) is set to run for multiple iterations, the consequent iterations will not be executed.
7. `ExitTestIteration`: This is used to abort the execution of the current test iteration. When a test is set to run for multiple iterations, control returns to the next iteration (the beginning of the first `Action` script of the test). If the iteration is the last, then the test will stop.
8. `GetLastError`: This is used to retrieve the last VBScript runtime error code (`Long Integer`). See the previous example with

`DescribeResult`.

9. `InvokeApplication`: This is obsolete; it is used to invoke an executable file. Use instead, the `SystemUtil.Run` method (see the previous sections in this chapter). The syntax is as follows:

```
InvokeApplication "iexplore.exe"
```

10. `LoadAndRunAction`: This is used to execute a reusable `Action` during runtime, without previously inserting a call and hence, associating it with the test. The syntax is similar to the `RunAction` statement, with one difference—one must supply the path of the test in which `Action` is stored:

```
LoadAndRunAction TestPath, ActionName, [iterations], [parameters]
```

11. The only advantage of this statement is that it enables a more flexible test flow. For example, one can envision that a mechanism within the name of the next action is determined based on the current context, namely, the result of a process and so on. Again, it is one of the very few statements that is highly recommended as not to be used. The risk is obvious, as was with the `ExecuteFile` statement; loading an `Action` in such a fashion exposes the run session to the risks of missing resources, such as an SOR, function library, or `DataSheet`.
12. `OptionalStep`: This is used to define a statement, which does not reflect in the results if it fails to execute. It is only to be used together with Test Objects, as shown in the following example:

```
OptionalStep.Dialog("Confirm Save As").WinButton("Yes").Click
```

This will try to click on the **Yes** button of **Dialog Confirm Save As**, but if the object does not exist, the test will continue to run, and no error message will be displayed.

13. `Print`: This is used to print messages, such as values of variables and so on, to the output pane. The syntax is as follows:

```
Print "MyString"

Print StringVariable
```

14. `RegisterUserFunc`: This is used to override Test Object methods. See the *Overriding a Test Object method* recipe (`RegisterUserFunc`) in Chapter 4, *Method Overriding*.
15. `RunAction`: This is used to launch the execution of an `Action`, whether internal or external to the test. If external, it must have association with the test beforehand using the add `Call` to `Action` via the UFT GUI. To launch the execution of unlinked reusable actions, see `LoadAndRunAction`. The syntax is as follows:

```
RunAction "Action2", "1-4", [param1, param2, …, param3]
SetLastError 9 'Assigns 9 to the runtime Err.Number
```

16. `UnregisterUserFunc`: This is used to remove the override from Test Object methods. See the *Overriding a Test Object method* recipe (`RegisterUserFunc`) in Chapter 4, *Method Overriding*.
17. `Wait`: This is used to slow down the script execution. It is recommended to use synchronization points with `WaitProperty` and `Exist`, because the wait time is not specific, and the script would still face the risk of failure. The syntax is as follows:

```
Wait seconds, milliseconds
```