How to Read Data into R Using the scan() Function

Authored by

stats writer

January 31, 2026

Use the scan() Function in R (With Example)

The scan() function in R is an exceptionally powerful, yet often overlooked, utility designed specifically for reading data from external sources. Unlike higher-level import functions like `read.csv()` or `read.table()`, scan() operates at a more fundamental level, allowing the user precise control over how input is parsed, tokenized, and structured within the R environment. This low-level approach makes it highly efficient, particularly when dealing with massive files or non-standard data layouts that might challenge conventional reading routines. It serves as a crucial bridge, enabling the conversion of raw text streams--whether from local files, database connections, or direct console input--into manageable R objects, typically a vector or a list.

The versatility of scan() stems from its ability to handle heterogeneous data formats. By allowing the user to explicitly define the expected data types and input structure--using parameters such as `what` and `nmax`--it adapts seamlessly to varying requirements. For instance, when importing a simple log file containing sequential timestamps and numerical values, the user

can optimize the process by specifying only numeric or character **vector** output. This inherent flexibility makes scan() an indispensable function for developers and analysts who frequently encounter bespoke or legacy data files that require careful, customized import procedures before manipulation and analysis can commence. Utilizing scan() effectively ensures that data integrity is maintained from the moment of import.

1. The Role of the scan() Function in R Data Import

While many users default to standard functions for importing well-structured delimited files, scan() excels in scenarios demanding granular control. Its primary role is to read items sequentially until the end of the file is reached, or until a predefined number of items (tokens) has been successfully read. This tokenization process is guided by the user-defined separator, which defaults to whitespace. This capability is especially beneficial when reading large datasets where speed is paramount, as scan() generally offers performance advantages over functions that must perform extensive type checking and column structuring during the initial read phase.

Consider a scenario where you are reading a simple, single-column file containing millions of experimental measurements. Using scan() to directly read this into a numeric vector bypasses the overhead associated with interpreting headers, handling quotes, and creating a temporary data frame structure. Furthermore, scan() is uniquely suited for interactive data input, allowing users to terminate input with an empty line--a feature rarely supported by other file reading utilities. This dual capability--high-performance file reading and flexible interactive data entry--solidifies its position as a fundamental tool in the R data management toolkit. This function is designed to read data from a file into either a vector or a list.

2. Detailed Syntax Breakdown: Mastering the scan() Arguments

To use scan() effectively, a comprehensive understanding of its syntax and primary arguments is essential. The basic syntax structure reflects its intent to sequentially read specified data types from an input source:

```
scan(file = "", what = double(), ...)
```

This structure reveals the three core components necessary for any call to scan(), though many other arguments exist to fine-tune the reading process. Understanding how these parameters interact dictates the success of the data import operation. It is worth noting that while the default settings are designed for general use, highly specific tasks require deep knowledge of optional arguments.

The core arguments are:

file: This argument specifies the location of the data source. Typically, this is a character string defining the path to a local file, such as `"C:/path/to/data.txt"`. The file argument is flexible; it can also accept a URL for remote data access, or the empty string `""`, which instructs scan() to read input interactively from the console. For production environments, specifying the exact file path is crucial for reproducibility.

what: Arguably the most critical argument, `what` defines the type and structure of the items to be read. If `what` is a single atomic **vector** type (e.g., `double()`, `integer()`, `character()`), scan() returns a single corresponding **vector**. If `what` is defined as a **list**, as demonstrated in the practical

example, scan() reads the file field-by-field, corresponding to the structure of the input list elements.

Refer to the official R documentation for a complete list of additional arguments you can use with the scan() function, such as `nmax` (maximum number of data items to read), `sep` (the field separator), and `skip` (number of lines to skip before reading).

3. Specifying Data Types: Utilizing the what Argument

The way scan() processes data is entirely dependent on the value assigned to the `what` argument. This argument acts as a template, guiding the function on how to interpret the raw tokens read from the input stream. When reading a file that contains uniform data--such as a file comprised entirely of floating-point numbers--the `what` argument should be set to `double()` (or `0.0`), ensuring that the output is a numeric vector optimized for numerical calculations. Similarly, `integer()` (or `0L`) is used for whole numbers, and `character()` (or `""`) for text strings.

However, when dealing with structured data, like a comma-separated values (CSV) file that contains mixed

data types (e.g., names, ages, salaries), the `what` argument must be defined as a **list**. Each element within this template **list** dictates the type of data expected for the corresponding field in the input file. For example, if the file has three columns--character, numeric, and integer--the `what` argument would be specified as `list("", 0.0, 0L)`. This ensures that the first column is read as text, the second as double-precision numbers, and the third as integers. The function then returns a **list** object where each element corresponds to a column from the input file.

If the `what` argument is omitted, scan() defaults to reading all input as character strings, equivalent to setting `what = character()`. While this provides maximum flexibility, it necessitates subsequent type coercion, which can slightly increase processing time and complexity. Therefore, explicitly defining the data types using `what` is considered best practice, particularly for large-scale data ingestion tasks where performance optimization is critical. The efficiency gained by pre-defining types outweighs the minor effort required to inspect the input file structure beforehand.

4. Practical Implementation: Setting up the Data Environment

Before executing the scan() command, it is vital to ensure the computing environment is correctly configured to locate and access the external data file. This involves verifying the file path and understanding the structure of the data itself. The file path must be absolute or relative to the current working directory in R. Incorrect path specification is the most frequent cause of errors when using file input functions.

For cross-platform compatibility, it is recommended to use forward slashes (/) in file paths, even on Windows systems, or to utilize the `file.path()` function in R, which automatically adjusts delimiters based on the operating system. In the context of our example, we are simulating a file located on a Windows desktop, using the syntax common in that environment.

In this demonstration, we will assume a basic comma-separated values (CSV) file structure, which is a common format for storing tabular data. While scan() is not typically the first choice for complex CSV files, it handles simple, clean delimited files exceptionally well. The key setup step here is to accurately define the

delimiters within the file; since our example uses commas, we will need to ensure the `sep` argument is correctly set, though for simplicity in the following example, we rely on scan()'s list parsing ability where separation is implied by the template structure.

Example: How to Use scan() Function in R

The following example shows how to use the scan() function in practice to handle structured data.

5. Defining the Data File Structure

Suppose I have a CSV file called data.csv saved in the following location:

C:UsersBobDesktopdata.csv

And suppose the CSV file contains the following data, including a header row:

team, points, assists
'A', 78, 12
'B', 85, 20
'C', 93, 23
'D', 90, 8
'E', 91, 14

To successfully import this data using scan(), we must first analyze the column types. We have one character column (`team`) and two numeric/integer columns (`points` and `assists`). This dictates the necessary structure for the `what` argument, which must be a list template reflecting these three types.

6. Reading Structured Data: A Concrete Example of scan() in Action

We utilize the list definition for the `what` argument to instruct scan() to read the file column-wise, distributing the tokens into three separate vectors within the final list object. Since we use `list("", "", "")`, the function interprets all input fields as character strings, including the numeric data and the header row tokens. This is the code required to quickly read the data from this file into R:

```
#read in data.csv into list
data <- scan("C:UsersBobDesktopdata.csv", what = list("", "", ""))

#view data
data

]
```

"team" "A" "B" "C" "D" "E"

]
"points" "78" "85" "93" "90" "91"

]
"assists" "12" "20" "23" "8" "14"

As demonstrated by the output, the scan() function has successfully read the data from the file into a list object named `data`. Each element of this list corresponds to a column in the original CSV file, with the header included as the first element of each resulting vector.

7. Handling Output and Verification: Lists vs. Vectors

When scan() is executed using a multi-element template in the `what` argument, the resulting R object is always a list. This structure allows for the retention of different data types (though in the example above, they were all coerced to character strings). Verifying the class of the resulting object is a necessary step to ensure the integrity of the import process.

We can verify that this object is a list by using the class() function:

```
#view class of data object
class(data)

"list"
```

The result confirms that `data` is stored as a list. To utilize this imported data for standard statistical operations, it would typically need to be converted into a data frame using the `as.data.frame()` function, often after cleaning the header row tokens that scan() included.

8. Conclusion: Optimizing Data Workflow with scan()

The scan() function represents a fundamental mechanism for data acquisition in R. Although more specialized functions exist for common file formats, scan() remains unmatched in its efficiency for reading large quantities of uniform data and its flexibility for handling non-standard input structures, including interactive console entries. By mastering the core arguments, particularly `file` and the type definition provided by `what`, users gain precise control over the data import process.

The following tutorials explain how to import other file

types into R: