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

Authored by

stats writer

January 15, 2026

The R programming language offers a variety of functions for data input, and among the most flexible and fundamental is the **scan()** function. This utility is designed to read data efficiently from external files or directly from user input, making it a crucial tool for data preparation and analysis. Unlike higher-level importing functions like `read.csv()`, **scan()** provides granular control over how data elements are parsed and returned, typically in the form of a vector or a list. For instance, if you possess a plain text file containing pure numeric data, utilizing the **scan()** function permits you to swiftly import this data into the R environment, enabling immediate performance of statistical operations, complex modeling, or sophisticated data visualization techniques. This function proves invaluable for seamlessly integrating data sourced from diverse origins and structuring it optimally for subsequent analysis.

Understanding the Core Purpose of the scan() Function

The primary utility of the **scan()** function lies in its ability to read individual data elements sequentially, treating the input as a stream. This contrasts sharply with functions like `read.table()`, which are optimized for reading rectangular datasets, often referred to as data frames. Because **scan()** focuses on raw input streams, it is exceptionally fast and highly flexible, particularly when dealing with unstructured data or when you only need to import specific parts of a file. It is often employed for reading large files element by element, offering performance advantages in high-volume data operations where speed is paramount.

When using **scan()**, the output structure is largely determined by the `what` argument. If `what` specifies a single type (e.g., `double()` for numeric data), the output will be a flat, homogeneous vector. If, however, `what` is provided as a list of types (e.g., `list("", 0, "")`), the function expects multiple fields per record and returns the data as a list, where each element corresponds to a column or field in the input file. This versatility makes **scan()** suitable for complex parsing tasks that require field separation but not immediate data frame creation.

Essential Syntax and Key Arguments

To effectively utilize **scan()**, mastering its fundamental syntax and critical arguments is necessary. This function provides precise control over the input process, allowing users to define exactly what is read and how it is interpreted. The basic function call is intuitive yet powerful, forming the foundation for more complex data ingestion tasks in R.

The basic syntax for the **scan()** function is structured as follows:

scan(file = "", what = double(), sep = "", quote = """", nmax = -1, quiet = FALSE, ...)

The most important arguments that determine the function's behavior are detailed below:

file: This argument specifies the source of the data. Typically, it is a character string containing the path and name of the file to be read. If left as the default `""`, **scan()** reads data from the standard input (usually the console), requiring the user to manually type the input and press Enter twice to signal the end of input.
what: Perhaps the most critical argument, `what` defines the type of data expected and, crucially, the structure of the resulting R object. It must be provided as an empty vector or a list corresponding to the data types in the file (e.g., `integer()`, `character()`, `double()`).
sep: This defines the field separator character. By default, **scan()** expects whitespace (spaces, tabs, newlines) to separate data fields. If the data is comma-separated (like a CSV file), you must explicitly set `sep = ","`.
nmax: An optional argument specifying the maximum number of data items to read. Setting `nmax` to a positive integer limits the scan, which is useful for reading only the first few records of a very large file for inspection, optimizing memory use.

For a complete and exhaustive reference on all available parameters and their advanced configurations, users should consult the official R documentation for the **scan()** function, as it contains many additional arguments for handling specialized input formats, such as skipping lines or managing quotes.

Practical Example: Importing Tabular Data

While **scan()** is generally excellent for reading homogeneous vectors, it can also be configured to import structured, heterogeneous data, such as records found in a CSV file. This example demonstrates how to leverage the `what` argument to instruct **scan()** to read multiple columns of differing types, resulting in a structured R list object. This approach bypasses standard table readers for direct control over data parsing.

Consider a scenario where an analyst needs to import a simple dataset detailing sports teams, their points, and assists. Suppose we have a comma-separated values file named **data.csv** stored at a specific location on the local file system. We must define the file path accurately, using appropriate escaping for the path separators:

C:UsersBobDesktopdata.csv

The content of the **data.csv** file includes a header row followed by five records of data. Notice that the data is delimited by commas:

team, points, assists
'A', 78, 12
'B', 85, 20
'C', 93, 23

'D', 90, 8
'E', 91, 14

To successfully read this data, we must account for the column separator (the comma) and the number of fields (three). We configure the **scan()** function accordingly using `sep = ","` and setting `what` as a list specifying three fields. By using empty strings in the `what` list, we instruct **scan()** to treat all three columns as character strings upon initial import, providing maximum flexibility before any type conversions are manually applied.

Executing the scan() Command in R

The following code snippet demonstrates the execution of the **scan()** function to quickly read the structured data from the specified CSV file into the R environment. Note the structure of the `what` argument: `list("", "", "")` indicates that **scan()** should read three separate fields per line. Since we are dealing with a Windows file path, the backslashes must be handled correctly in the R character string.

 # Read data.csv into a list structure, specifying the comma separator
data <- scan("C:UsersBobDesktopdata.csv", what = list("", "", ""), sep = ",")

View the resulting R object
data

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

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

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

Interpreting the Resulting List Object

Upon execution, the **scan()** function has successfully read the data, including the header row, and organized it into an R list structure. This list, named `data`, contains three elements, where each element corresponds to one column from the input file. List element `]` holds the 'team' data, `]` holds 'points' data, and `]` holds 'assists' data. It is important to notice that because we specified `what = list("", "", "")`, all values--including numerical values like '78', '85', etc.--were imported as character strings. This is a crucial distinction when working with **scan()** versus data

frame readers.

The resulting list structure is mandatory here; **scan()** returns a list when the `what` argument is a list object itself, ensuring that the columnar structure of the data is preserved, facilitating subsequent cleaning and conversion steps, such as transforming the list into a data frame using functions like `as.data.frame()`.

Verifying the Data Structure using class()

To confirm that the data object created by **scan()** is indeed a list, we can use the fundamental **class()** function in R. This confirmation step is standard practice after data importation to ensure the object is ready for subsequent operations that expect a list input, preventing runtime errors due to mismatched object types.

 # View class of the data object
class(data)

"list"

The output `"list"` confirms the successful creation of the desired data structure. Had we provided only a single data type in the `what` argument (e.g., `what = character()`) and the input file contained only one sequence of elements, the resulting object would have been a single vector.

Advanced Applications and Considerations

While **scan()** is powerful, users must be mindful of its default behaviors, especially regarding data type coercion and separators. By default, **scan()** treats all consecutive whitespace characters as a single delimiter. If your data uses fixed-width fields or requires complex parsing rules, you may need to utilize the `text` argument or functions from the `stringr` package to pre-process the data before scanning.

One key consideration for production environments is handling missing values. By default, **scan()** reads empty fields based on the type specified in `what`, which may lead to unintended results if missing values (NA) are expected. Users can utilize the `na.strings` argument within **scan()** to explicitly define which character strings in the input file (e.g., `"NA"`, `"NULL"`, or a blank space) should be treated as missing values, significantly improving the reliability and statistical integrity of the imported data.

Alternative Data Import Methods in R

While **scan()** is a fundamental and robust method for reading raw data, R offers other, often

simpler, methods tailored for specific file structures. Understanding these alternatives helps in selecting the most efficient tool for any given task, balancing control versus convenience.

If the file is a standard rectangular dataset (like a typical CSV file or tab-delimited file), functions from the `readr` package (like `read_csv()`) or base R functions (like `read.table()` or `read.csv()`) are often preferred because they automatically handle headers, row names, and data type inference, typically returning a data frame directly without requiring manual list-to-data frame conversion. These functions are generally optimized for speed and user experience for common data formats.

However, **scan()** remains the function of choice when:

Performance is critical for reading a massive file, and you only need a raw vector (e.g., reading only one column of a vast file).
The file structure is highly irregular, containing comments, blank lines, or unusual delimiters that require explicit, element-level control over the reading process.
You need to read data directly from the console, an internal connection object, or a streaming API rather than a persistent, structured file on disk.

The following tutorials explain how to import other file types into R: