
Data Structures

Introduction

You can get pretty far in R just using vectors. That's what [Chapter 2](#) is all about. This chapter moves beyond vectors to recipes for matrices, lists, factors, and data frames. If you have preconceptions about data structures, I suggest you put them aside. R does data structures differently.

If you want to study the technical aspects of R's data structures, I suggest reading [R in a Nutshell](#) (O'Reilly) and the *R Language Definition*. My notes here are more informal. These are things I wish I'd known when I started using R.

Vectors

Here are some key properties of vectors:

Vectors are homogeneous

All elements of a vector must have the same type or, in R terminology, the same mode.

Vectors can be indexed by position

So `v[2]` refers to the second element of `v`.

Vectors can be indexed by multiple positions, returning a subvector

So `v[c(2,3)]` is a subvector of `v` that consists of the second and third elements.

Vector elements can have names

Vectors have a `names` property, the same length as the vector itself, that gives names to the elements:

```
> v <- c(10, 20, 30)
> names(v) <- c("Moe", "Larry", "Curly")
> print(v)
Moe Larry Curly
 10   20   30
```

If vector elements have names then you can select them by name
Continuing the previous example:

```
> v["Larry"]  
Larry  
20
```

Lists

Lists are heterogeneous

Lists can contain elements of different types; in R terminology, list elements may have different modes. Lists can even contain other structured objects, such as lists and data frames; this allows you to create recursive data structures.

Lists can be indexed by position

So `lst[[2]]` refers to the second element of `lst`. Note the double square brackets.

Lists let you extract sublists

So `lst[c(2,3)]` is a sublist of `lst` that consists of the second and third elements. Note the single square brackets.

List elements can have names

Both `lst[["Moe"]]` and `lst$Moe` refer to the element named “Moe”.

Since lists are heterogeneous and since their elements can be retrieved by name, a list is like a *dictionary* or *hash* or *lookup table* in other programming languages ([Recipe 5.9](#)). What’s surprising (and cool) is that in R, unlike most of those other programming languages, lists can also be indexed by position.

Mode: Physical Type

In R, every object has a *mode*, which indicates how it is stored in memory: as a number, as a character string, as a list of pointers to other objects, as a function, and so forth:

Object	Example	Mode
Number	3.1415	numeric
Vector of numbers	<code>c(2.7.182, 3.1415)</code>	numeric
Character string	"Moe"	character
Vector of character strings	<code>c("Moe", "Larry", "Curly")</code>	character
Factor	<code>factor(c("NY", "CA", "IL"))</code>	numeric
List	<code>list("Moe", "Larry", "Curly")</code>	list
Data frame	<code>data.frame(x=1:3, y=c("NY", "CA", "IL"))</code>	list
Function	<code>print</code>	function

The `mode` function gives us this information:

```

> mode(3.1415)                # Mode of a number
[1] "numeric"
> mode(c(2.7182, 3.1415))     # Mode of a vector of numbers
[1] "numeric"
> mode("Moe")                 # Mode of a character string
[1] "character"
> mode(list("Moe", "Larry", "Curly")) # Mode of a list
[1] "list"

```

A critical difference between a vector and a list can be summed up this way:

- In a vector, all elements must have the same mode.
- In a list, the elements can have different modes.

Class: Abstract Type

In R, every object also has a *class*, which defines its abstract type. The terminology is borrowed from object-oriented programming. A single number could represent many different things: a distance, a point in time, a weight. All those objects have a mode of “numeric” because they are stored as a number; but they could have different classes to indicate their interpretation.

For example, a `Date` object consists of a single number:

```

> d <- as.Date("2010-03-15")
> mode(d)
[1] "numeric"
> length(d)
[1] 1

```

But it has a class of `Date`, telling us how to interpret that number; namely, as the number of days since January 1, 1970:

```

> class(d)
[1] "Date"

```

R uses an object’s class to decide how to process the object. For example, the generic function `print` has specialized versions (called *methods*) for printing objects according to their class: `data.frame`, `Date`, `lm`, and so forth. When you print an object, R calls the appropriate `print` function according to the object’s class.

Scalars

The quirky thing about scalars is their relationship to vectors. In some software, scalars and vectors are two different things. In R, they are the same thing: a scalar is simply a vector that contains exactly one element. In this book I often use the term “scalar”, but that’s just shorthand for “vector with one element.”

Consider the built-in constant `pi`. It is a scalar:

```

> pi
[1] 3.141593

```

Since a scalar is a one-element vector, you can use vector functions on `pi`:

```
> length(pi)
[1] 1
```

You can index it. The first (and only) element is π , of course:

```
> pi[1]
[1] 3.141593
```

If you ask for the second element, there is none:

```
> pi[2]
[1] NA
```

Matrices

In R, a matrix is just a vector that has dimensions. It may seem strange at first, but you can transform a vector into a matrix simply by giving it dimensions.

A vector has an attribute called `dim`, which is initially `NULL`, as shown here:

```
> A <- 1:6
> dim(A)
NULL
> print(A)
[1] 1 2 3 4 5 6
```

We give dimensions to the vector when we set its `dim` attribute. Watch what happens when we set our vector dimensions to 2×3 and print it:

```
> dim(A) <- c(2,3)
> print(A)
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Voilà! The vector was reshaped into a 2×3 matrix.

A matrix can be created from a list, too. Like a vector, a list has a `dim` attribute, which is initially `NULL`:

```
> B <- list(1,2,3,4,5,6)
> dim(B)
NULL
```

If we set the `dim` attribute, it gives the list a shape:

```
> dim(B) <- c(2,3)
> print(B)
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Voilà! We have turned this list into a 2×3 matrix.

Arrays

The discussion of matrices can be generalized to 3-dimensional or even n -dimensional structures: just assign more dimensions to the underlying vector (or list). The following example creates a 3-dimensional array with dimensions $2 \times 3 \times 2$:

```
> D <- 1:12
> dim(D) <- c(2,3,2)
> print(D)
, , 1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
, , 2
      [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Note that R prints one “slice” of the structure at a time, since it’s not possible to print a 3-dimensional structure on a 2-dimensional medium.

Matrices Made from Lists

It strikes me as very odd that we can turn a list into a matrix just by giving the list a `dim` attribute. But wait; it gets stranger.

Recall that a list can be heterogeneous (mixed modes). We can start with a heterogeneous list, give it dimensions, and thus create a heterogeneous matrix. This code snippet creates a matrix that is a mix of numeric and character data:

```
> C <- list(1, 2, 3, "X", "Y", "Z")
> dim(C) <- c(2,3)
> print(C)
      [,1] [,2] [,3]
[1,]    1    3  "Y"
[2,]    2  "X"  "Z"
```

To me this is strange because I ordinarily assume a matrix is purely numeric, not mixed. R is not that restrictive.

The possibility of a heterogeneous matrix may seem powerful and strangely fascinating. However, it creates problems when you are doing normal, day-to-day stuff with matrices. For example, what happens when the matrix C (above) is used in matrix multiplication? What happens if it is converted to a data frame? The answer is that odd things happen.

In this book, I generally ignore the pathological case of a heterogeneous matrix. I assume you’ve got simple, vanilla matrices. Some recipes involving matrices may work oddly (or not at all) if your matrix contains mixed data. Converting such a matrix to a vector or data frame, for instance, can be problematic ([Recipe 5.33](#)).

Factors

A factor looks like a vector, but it has special properties. R keeps track of the unique values in a vector, and each unique value is called a *level* of the associated factor. R uses a compact representation for factors, which makes them efficient for storage in data frames. In other programming languages, a factor would be represented by a *vector of enumerated values*.

There are two key uses for factors:

Categorical variables

A factor can represent a categorical variable. Categorical variables are used in contingency tables, linear regression, analysis of variance (ANOVA), logistic regression, and many other areas.

Grouping

This is a technique for labeling or tagging your data items according to their group. See the “[Introduction](#)” to [Chapter 6](#).

Data Frames

A data frame is powerful and flexible structure. Most serious R applications involve data frames. A data frame is intended to mimic a dataset, such as one you might encounter in SAS or SPSS.

A data frame is a tabular (rectangular) data structure, which means that it has rows and columns. It is not implemented by a matrix, however. Rather, a data frame is a list:

- The elements of the list are vectors and/or factors.*
- Those vectors and factors are the columns of the data frame.
- The vectors and factors must all have the same length; in other words, all columns must have the same height.
- The equal-height columns give a rectangular shape to the data frame.
- The columns must have names.

Because a data frame is both a list and a rectangular structure, R provides two different paradigms for accessing its contents:

- You can use list operators to extract columns from a data frame, such as `dfrm[i]`, `dfrm[[i]]`, or `dfrm$name`.
- You can use matrix-like notation, such as `dfrm[i,j]`, `dfrm[i,]`, or `dfrm[,j]`.

Your perception of a data frame likely depends on your background:

* A data frame can be built from a mixture of vectors, factors, and matrices. The columns of the matrices become columns in the data frame. The number of *rows* in each matrix must match the *length* of the vectors and factors. In other words, all elements of a data frame must have the same *height*.

To a statistician

A data frame is a table of observations. Each row contains one observation. Each observation must contain the same variables. These variables are called columns, and you can refer to them by name. You can also refer to the contents by row number and column number, just as with a matrix.

To a SQL programmer

A data frame is a table. The table resides entirely in memory, but you can save it to a flat file and restore it later. You needn't declare the column types because R figures that out for you.

To an Excel user

A data frame is like a worksheet, or perhaps a range within a worksheet. It is more restrictive, however, in that each column has a type.

To an SAS user

A data frame is like a SAS dataset for which all the data resides in memory. R can read and write the data frame to disk, but the data frame must be in memory while R is processing it.

To an R programmer

A data frame is a hybrid data structure, part matrix and part list. A column can contain numbers, character strings, or factors but not a mix of them. You can index the data frame just like you index a matrix. The data frame is also a list, where the list elements are the columns, so you can access columns by using list operators.

To a computer scientist

A data frame is a rectangular data structure. The columns are strongly typed, and each column must be numeric values, character strings, or a factor. Columns must have labels; rows may have labels. The table can be indexed by position, column name, and/or row name. It can also be accessed by list operators, in which case R treats the data frame as a list whose elements are the columns of the data frame.

To an executive

You can put names and numbers into a data frame. It's easy! A data frame is like a little database. Your staff will enjoy using data frames.

5.1 Appending Data to a Vector

Problem

You want to append additional data items to a vector.

Solution

Use the vector constructor `c()` to construct a vector with the additional data items:

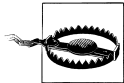
```
> v <- c(v,newItems)
```

For a single item, you can also assign the new item to the next vector element. R will automatically extend the vector:

```
> v[length(v)+1] <- newItem
```

Discussion

If you ask me about appending a data item to a vector, I will suggest that maybe you shouldn't.



R works best when you think about entire vectors, not single data items. Are you repeatedly appending items to a vector? If so, then you are probably working inside a loop. That's OK for small vectors, but for large vectors your program will run slowly. The memory management in R works poorly when you repeatedly extend a vector by one element. Try to replace that loop with vector-level operations. You'll write less code, and R will run much faster.

Nonetheless, one does occasionally need to append data to vectors. My experiments show that the most efficient way is to create a new vector using the vector constructor (c) to join the old and new data. This works for appending single elements or multiple elements:

```
> v <- c(1,2,3)
> v <- c(v,4)      # Append a single value to v
> v
[1] 1 2 3 4
> w <- c(5,6,7,8)
> v <- c(v,w)      # Append an entire vector to v
> v
[1] 1 2 3 4 5 6 7 8
```

You can also append an item by assigning it to the position past the end of the vector, as shown in the Solution. In fact, R is very liberal about extending vectors. You can assign to any element and R will expand the vector to accommodate your request:

```
> v <- c(1,2,3)      # Create a vector of three elements
> v[10] <- 10        # Assign to the 10th element
> v                  # R extends the vector automatically
[1] 1 2 3 NA NA NA NA NA 10
```

Note that R did not complain about the out-of-bounds subscript. It just extended the vector to the needed length, filling with NA.

R includes an `append` function that creates a new vector by appending items to an existing vector. However, my experiments show that this function runs more slowly than both the vector constructor and the element assignment.

5.2 Inserting Data into a Vector

Problem

You want to insert one or more data items into a vector.

Solution

Despite its name, the `append` function inserts data into a vector by using the `after` parameter, which gives the insertion point for the new item or items:

```
> append(vec, newvalues, after=n)
```

Discussion

The new items will be inserted at the position given by `after`. This example inserts 99 into the middle of a sequence:

```
> append(1:10, 99, after=5)
[1] 1 2 3 4 5 99 6 7 8 9 10
```

The special value of `after=0` means insert the new items at the head of the vector:

```
> append(1:10, 99, after=0)
[1] 99 1 2 3 4 5 6 7 8 9 10
```

The comments in [Recipe 5.1](#) apply here, too. If you are inserting single items into a vector, you might be working at the element level when working at the vector level would be easier to code and faster to run.

5.3 Understanding the Recycling Rule

Problem

You want to understand the mysterious Recycling Rule that governs how R handles vectors of unequal length.

Discussion

When you do vector arithmetic, R performs element-by-element operations. That works well when both vectors have the same length: R pairs the elements of the vectors and applies the operation to those pairs.

But what happens when the vectors have unequal lengths?

In that case, R invokes the Recycling Rule. It processes the vector element in pairs, starting at the first elements of both vectors. At a certain point, the shorter vector is exhausted while the longer vector still has unprocessed elements. R returns to the beginning of the shorter vector, “recycling” its elements; continues taking elements from

the longer vector; and completes the operation. It will recycle the shorter-vector elements as often as necessary until the operation is complete.

It's useful to visualize the Recycling Rule. Here is a diagram of two vectors, 1:6 and 1:3:

1:6	1:3
1	1
2	2
3	3
4	
5	
6	

Obviously, the 1:6 vector is longer than the 1:3 vector. If we try to add the vectors using $(1:6) + (1:3)$, it appears that 1:3 has too few elements. However, R recycles the elements of 1:3, pairing the two vectors like this and producing a six-element vector:

1:6	1:3	(1:6) + (1:3)
1	1	2
2	2	4
3	3	6
4	1	5
5	2	7
6	3	9

Here is what you see at the command line:

```
> (1:6) + (1:3)
[1] 2 4 6 5 7 9
```

It's not only vector operations that invoke the Recycling Rule; functions can, too. The `cbind` function can create column vectors, such as the following column vectors of 1:6 and 1:3. The two column have different heights, of course:

```
> cbind(1:6)
[,1]
[1,] 1
[2,] 2
[3,] 3
[4,] 4
[5,] 5
[6,] 6
```

```
> cbind(1:3)
      [,1]
[1,]    1
[2,]    2
[3,]    3
```

If we try binding these column vectors together into a two-column matrix, the lengths are mismatched. The 1:3 vector is too short, so `cbind` invokes the Recycling Rule and recycles the elements of 1:3:

```
> cbind(1:6, 1:3)
      [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    1
[5,]    5    2
[6,]    6    3
```

If the longer vector's length is not a multiple of the shorter vector's length, R gives a warning. That's good, since the operation is highly suspect and there is likely a bug in your logic:

```
> (1:6) + (1:5)      # Oops! 1:5 is one element too short
[1]  2  4  6  8 10  7
Warning message:
In (1:6) + (1:5) :
  longer object length is not a multiple of shorter object length
```

Once you understand the Recycling Rule, you will realize that operations between a vector and a scalar are simply applications of that rule. In this example, the 10 is recycled repeatedly until the vector addition is complete:

```
> (1:6) + 10
[1] 11 12 13 14 15 16
```

5.4 Creating a Factor (Categorical Variable)

Problem

You have a vector of character strings or integers. You want R to treat them as a factor, which is R's term for a categorical variable.

Solution

The `factor` function encodes your vector of discrete values into a factor:

```
> f <- factor(v)      # v is a vector of strings or integers
```

If your vector contains only a subset of possible values and not the entire universe, then include a second argument that gives the possible levels of the factor:

```
> f <- factor(v, levels)
```

Discussion

In R, each possible value of a categorical variable is called a *level*. A vector of levels is called a *factor*. Factors fit very cleanly into the vector orientation of R, and they are used in powerful ways for processing data and building statistical models.

Most of the time, converting your categorical data into a factor is a simple matter of calling the `factor` function, which identifies the distinct levels of the categorical data and packs them into a factor:

```
> f <- factor(c("Win", "Win", "Lose", "Tie", "Win", "Lose"))
> f
[1] Win Win Lose Tie Win Lose
Levels: Lose Tie Win
```

Notice that when we printed the factor, `f`, R did not put quotes around the values. They are levels, not strings. Also notice that when we printed the factor, R also displayed the distinct levels below the factor.

If your vector contains only a subset of all the possible levels, then R will have an incomplete picture of the possible levels. Suppose you have a string-valued variable `wday` that gives the day of the week on which your data was observed:

```
> f <- factor(wday)
> f
[1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Thu Tue Wed
```

R thinks that Monday, Thursday, Tuesday, and Wednesday are the only possible levels. Friday is not listed. Apparently, the lab staff never made observations on Friday, so R does not know that Friday is a possible value. Hence you need to list the possible levels of `wday` explicitly:

```
> f <- factor(wday, c("Mon", "Tue", "Wed", "Thu", "Fri"))
> f
[1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Tue Wed Thu Fri
```

Now R understands that `f` is a factor with five possible levels. It knows their correct order, too. It originally put Thursday before Tuesday because it assumes alphabetical order by default.[†] The explicit second argument defines the correct order.

In many situations it is not necessary to call `factor` explicitly. When an R function requires a factor, it usually converts your data to a factor automatically. The `table` function, for instance, works only on factors, so it routinely converts its inputs to factors without asking. You must explicitly create a factor variable when you want to specify the full set of levels or when you want to control the ordering of levels.

[†] More precisely, it orders the names according to your Locale.

See Also

See [Recipe 12.6](#) to create a factor from continuous data.

5.5 Combining Multiple Vectors into One Vector and a Factor

Problem

You have several groups of data, with one vector for each group. You want to combine the vectors into one large vector and simultaneously create a parallel factor that identifies each value's original group.

Solution

Create a list that contains the vectors. Use the `stack` function to combine the list into a two-column data frame:

```
> comb <- stack(list(v1=v1, v2=v2, v3=v3))    # Combine 3 vectors
```

The data frame's columns are called `values` and `ind`. The first column contains the data, and the second column contains the parallel factor.

Discussion

Why in the world would you want to mash all your data into one big vector and a parallel factor? The reason is that many important statistical functions require the data in that format.

Suppose you survey freshmen, sophomores, and juniors regarding their confidence level ("What percentage of the time do you feel confident in school?"). Now you have three vectors, called `freshmen`, `sophomores`, and `juniors`. You want to perform an ANOVA analysis of the differences between the groups. The ANOVA function, `aov`, requires one vector with the survey results as well as a parallel factor that identifies the group. You can combine the groups using the `stack` function:

```
> comb <- stack(list(fresh=freshmen, soph=sophomores, jrs=juniors))
> print(comb)
  values ind
1  0.60 fresh
2  0.35 fresh
3  0.44 fresh
4  0.62 fresh
5  0.60 fresh
6  0.70 soph
7  0.61 soph
8  0.63 soph
9  0.87 soph
10 0.85 soph
11 0.70 soph
12 0.64 soph
```

```
13  0.76  jrs
14  0.71  jrs
15  0.92  jrs
16  0.87  jrs
```

Now you can perform the ANOVA analysis on the two columns:

```
> aov(values ~ ind, data=comb)
```

When building the list we must provide tags for the list elements (the tags are `fresh`, `soph`, and `jrs` in this example). Those tags are required because `stack` uses them as the levels of the parallel factor.

5.6 Creating a List

Problem

You want to create and populate a list.

Solution

To create a list from individual data items, use the `list` function:

```
> lst <- list(x, y, z)
```

Discussion

Lists can be quite simple, such as this list of three numbers:

```
> lst <- list(0.5, 0.841, 0.977)
> lst
[[1]]
[1] 0.5

[[2]]
[1] 0.841

[[3]]
[1] 0.977
```

When R prints the list, it identifies each list element by its position (`[[1]]`, `[[2]]`, `[[3]]`) and prints the element's value (e.g., `[1] 0.5`) under its position.

More usefully, lists can—unlike vectors—contain elements of different modes (types). Here is an extreme example of a mongrel created from a scalar, a character string, a vector, and a function:

```
> lst <- list(3.14, "Moe", c(1,1,2,3), mean)
> lst
[[1]]
[1] 3.14

[[2]]
```

```
[1] "Moe"

[[3]]
[1] 1 1 2 3

[[4]]
function (x, ...)
  UseMethod("mean")
<environment: namespace:base>
```

You can also build a list by creating an empty list and populating it. Here is our “mongrel” example built in that way:

```
> lst <- list()
> lst[[1]] <- 3.14
> lst[[2]] <- "Moe"
> lst[[3]] <- c(1,1,2,3)
> lst[[4]] <- mean
```

List elements can be named. The `list` function lets you supply a name for every element:

```
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
> lst
$mid
[1] 0.5

$right
[1] 0.841

$far.right
[1] 0.977
```

See Also

See the “[Introduction](#)” to this chapter for more about lists; see [Recipe 5.9](#) for more about building and using lists with named elements.

5.7 Selecting List Elements by Position

Problem

You want to access list elements by position.

Solution

Use one of these ways. Here, `lst` is a list variable:

```
lst[[n]]
  Select the  $n$ th element from the list.

lst[c(n1, n2, ..., nk)]
  Returns a list of elements, selected by their positions.
```

Note that the first form returns a single element and the second returns a list.

Discussion

Suppose we have a list of four integers, called `years`:

```
> years <- list(1960, 1964, 1976, 1994)
> years
[[1]]
[1] 1960

[[2]]
[1] 1964

[[3]]
[1] 1976

[[4]]
[1] 1994
```

We can access single elements using the double-square-bracket syntax:

```
> years[[1]]
[1] 1960
```

We can extract sublists using the single-square-bracket syntax:

```
> years[c(1,2)]
[[1]]
[1] 1960

[[2]]
[1] 1964
```

This syntax can be confusing because of a subtlety: there is an important difference between `lst[[n]]` and `lst[n]`. They are not the same thing:

`lst[[n]]`

This is an element, not a list. It is the *n*th element of `lst`.

`lst[n]`

This is a list, not an element. The list contains one element, taken from the *n*th element of `lst`. This is a special case of `lst[c(n1, n2, ..., nk)]` in which we eliminated the `c(...)` construct because there is only one *n*.

The difference becomes apparent when we inspect the structure of the result—one is a number; the other is a list:

```
> class(years[[1]])
[1] "numeric"

> class(years[1])
[1] "list"
```


The difference becomes annoyingly apparent when we `cat` the value. Recall that `cat` can print atomic values or vectors but complains about printing structured objects:

```
> cat(years[[1]], "\n")
1960
> cat(years[1], "\n")
Error in cat(list(...), file, sep, fill, labels, append) :
  argument 1 (type 'list') cannot be handled by 'cat'
```

We got lucky here because R alerted us to the problem. In other contexts, you might work long and hard to figure out that you accessed a sublist when you wanted an element, or vice versa.

5.8 Selecting List Elements by Name

Problem

You want to access list elements by their names.

Solution

Use one of these forms. Here, `lst` is a list variable:

```
lst[["name"]]
```

Selects the element called *name*. Returns `NULL` if no element has that name.

```
lst$name
```

Same as previous, just different syntax.

```
lst[c(name1, name2, ..., namek)]
```

Returns a list built from the indicated elements of `lst`.

Note that the first two forms return an element whereas the third form returns a list.

Discussion

Each element of a list can have a name. If named, the element can be selected by its name. This assignment creates a list of four named integers:

```
> years <- list(Kennedy=1960, Johnson=1964, Carter=1976, Clinton=1994)
```

These next two expressions return the same value—namely, the element that is named “Kennedy”:

```
> years[["Kennedy"]]
[1] 1960
> years$Kennedy
[1] 1960
```

The following two expressions return sublists extracted from `years`:

```
> years[c("Kennedy", "Johnson")]
$Kennedy
```

```
[1] 1960
```

```
$Johnson
```

```
[1] 1964
```

```
> years["Carter"]
```

```
$Carter
```

```
[1] 1976
```

Just as with selecting list elements by position ([Recipe 5.7](#)), there is an important difference between `lst[["name"]]` and `lst["name"]`. They are not the same:

```
lst[["name"]]
```

This is an element, not a list.

```
lst["name"]
```

This is a list, not an element. This is a special case of `lst[c(name1, name2, ..., namek)]` in which we don't need the `c(...)` construct because there is only one name.

See Also

See [Recipe 5.7](#) to access elements by position rather than by name.

5.9 Building a Name/Value Association List

Problem

You want to create a list that associates names and values—as would a dictionary, hash, or lookup table in another programming language.

Solution

The `list` function lets you give names to elements, creating an association between each name and its value:

```
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
```

If you have parallel vectors of names and values, you can create an empty list and then populate the list by using a vectorized assignment statement:

```
> lst <- list()
> lst[names] <- values
```

Discussion

Each element of a list can be named, and you can retrieve list elements by name. This gives you a basic programming tool: the ability to associate names with values.

You can assign element names when you build the list. The `list` function allows arguments of the form *name=value*:

```

> lst <- list(
+     far.left=0.023,
+     left=0.159,
+     mid=0.500,
+     right=0.841,
+     far.right=0.977)
> lst
$far.left
[1] 0.023

$left
[1] 0.159

$mid
[1] 0.5

$right
[1] 0.841

$far.right
[1] 0.977

```

One way to name the elements is to create an empty list and then populate it via assignment statements:

```

> lst <- list()
> lst$far.left <- 0.023
> lst$left <- 0.159
> lst$mid <- 0.500
> lst$right <- 0.841
> lst$far.right <- 0.977

```

Sometimes you have a vector of names and a vector of corresponding values:

```

> values <- pnorm(-2:2)
> names <- c("far.left", "left", "mid", "right", "far.right")

```

You can associate the names and the values by creating an empty list and then populating it with a vectorized assignment statement:

```

> lst <- list()
> lst[names] <- values
> lst
$far.left
[1] 0.02275013

$left
[1] 0.1586553

$mid
[1] 0.5

$right
[1] 0.8413447

$far.right
[1] 0.9772499

```

Once the association is made, the list can “translate” names into values through a simple list lookup:

```
> cat("The left limit is", lst[["left"]], "\n")
The left limit is 0.1586553
> cat("The right limit is", lst[["right"]], "\n")
The right limit is 0.8413447

> for (nm in names(lst)) cat("The", nm, "limit is", lst[[nm]], "\n")
The far.left limit is 0.02275013
The left limit is 0.1586553
The mid limit is 0.5
The right limit is 0.8413447
The far.right limit is 0.9772499
```

5.10 Removing an Element from a List

Problem

You want to remove an element from a list.

Solution

Assign `NULL` to the element. R will remove it from the list.

Discussion

To remove a list element, select it by position or by name, and then assign `NULL` to the selected element:

```
> years
$Kennedy
[1] 1960

$Johnson
[1] 1964

$Carter
[1] 1976

$Clinton
[1] 1994

> years[["Johnson"]] <- NULL           # Remove the element labeled "Johnson"
> years
$Kennedy
[1] 1960

$Carter
[1] 1976

$Clinton
[1] 1994
```

You can remove multiple elements this way, too:

```
> years[c("Carter", "Clinton")] <- NULL      # Remove two elements
> years
$Kennedy
[1] 1960
```

5.11 Flatten a List into a Vector

Problem

You want to flatten all the elements of a list into a vector.

Solution

Use the `unlist` function.

Discussion

There are many contexts that require a vector. Basic statistical functions work on vectors but not on lists, for example. If `iq.scores` is a list of numbers, then we cannot directly compute their mean:

```
> mean(iq.scores)
[1] NA
Warning message:
In mean.default(iq.scores) :
  argument is not numeric or logical: returning NA
```

Instead, we must flatten the list into a vector using `unlist` and then compute the mean of the result:

```
> mean(unlist(iq.scores))
[1] 106.4452
```

Here is another example. We can `cat` scalars and vectors, but we cannot `cat` a list:

```
> cat(iq.scores, "\n")
Error in cat(list(...), file, sep, fill, labels, append) :
  argument 1 (type 'list') cannot be handled by 'cat'
```

One solution is to flatten the list into a vector before printing:

```
> cat("IQ Scores:", unlist(iq.scores), "\n")
IQ Scores: 89.73383 116.5565 113.0454
```

See Also

Conversions such as this are discussed more fully in [Recipe 5.33](#).

5.12 Removing NULL Elements from a List

Problem

Your list contains NULL values. You want to remove them.

Solution

Suppose `lst` is a list some of whose elements are NULL. This expression will remove the NULL elements:

```
> lst[sapply(lst, is.null)] <- NULL
```

Discussion

Finding and removing NULL elements from a list is surprisingly tricky. I wrote the following expression after trying several other ways, including the obvious ones, and failing. Here's how it works:

1. R calls `sapply` to apply the `is.null` function to every element of the list.
2. `sapply` returns a vector of logical values that are TRUE wherever the corresponding list element is NULL.
3. R selects values from the list according to that vector.
4. R assigns NULL to the selected items, removing them from the list.

The curious reader may be wondering how a list can contain NULL elements, given that we remove elements by setting them to NULL ([Recipe 5.10](#)). The answer is that we can create a list containing NULL elements:

```
> lst <- list("Moe", NULL, "Curly")      # Create list with NULL element
> lst
[[1]]
[1] "Moe"

[[2]]
NULL

[[3]]
[1] "Curly"

> lst[sapply(lst, is.null)] <- NULL      # Remove NULL element from list
> lst
[[1]]
[1] "Moe"

[[2]]
[1] "Curly"
```

See Also

See [Recipe 5.10](#) for how to remove list elements.

5.13 Removing List Elements Using a Condition

Problem

You want to remove elements from a list according to a conditional test, such as removing elements that are negative or smaller than some threshold.

Solution

Build a logical vector based on the condition. Use the vector to select list elements and then assign `NULL` to those elements. This assignment, for example, removes all negative value from `lst`:

```
> lst[lst < 0] <- NULL
```

Discussion

This recipe is based on two useful features of R. First, a list can be indexed by a logical vector. Wherever the vector element is `TRUE`, the corresponding list element is selected. Second, you can remove a list element by assigning `NULL` to it.

Suppose we want to remove elements from `lst` whose value is zero. We construct a logical vector which identifies the unwanted values (`lst == 0`). Then we select those elements from the list and assign `NULL` to them:

```
> lst[lst == 0] <- NULL
```

This expression will remove `NA` values from the list:

```
> lst[is.na(lst)] <- NULL
```

So far, so good. The problems arise when you cannot easily build the logical vector. That often happens when you want to use a function that cannot handle a list. Suppose you want to remove list elements whose absolute value is less than 1. The `abs` function will not handle a list, unfortunately:

```
> lst[abs(lst) < 1] <- NULL
Error in abs(lst) : non-numeric argument to function
```

The simplest solution is flattening the list into a vector by calling `unlist` and then testing the vector:

```
> lst[abs(unlist(lst)) < 1] <- NULL
```

A more elegant solution uses `lapply` to apply the function to every element of the list:

```
> lst[lapply(lst, abs) < 1] <- NULL
```

Lists can hold complex objects, too, not just atomic values. Suppose that `mods` is a list of linear models created by the `lm` function. This expression will remove any model whose R^2 value is less than 0.30:

```
> mods[apply(mods, function(m) summary(m)$r.squared < 0.3)] <- NULL
```

See Also

See Recipes [5.7](#), [5.10](#), [5.11](#), [6.2](#), and [11.1](#).

5.14 Initializing a Matrix

Problem

You want to create a matrix and initialize it from given values.

Solution

Capture the data in a vector or list, and then use the `matrix` function to shape the data into a matrix. This example shapes a vector into a 2×3 matrix (i.e., two rows and three columns):

```
> matrix(vec, 2, 3)
```

Discussion

Suppose we want to create and initialize a 2×3 matrix. We can capture the initial data inside a vector and then shape it using the `matrix` function:

```
> theData <- c(1.1, 1.2, 2.1, 2.2, 3.1, 3.2)
> mat <- matrix(theData, 2, 3)
> mat
      [,1] [,2] [,3]
[1,]  1.1  2.1  3.1
[2,]  1.2  2.2  3.2
```

The first argument of `matrix` is the data, the second argument is the number of rows, and the third argument is the number of columns. Observe that the matrix was filled column by column, not row by row.

It's common to initialize an entire matrix to one value such as zero or NA. If the first argument of `matrix` is a single value, then R will apply the Recycling Rule and automatically replicate the value to fill the entire matrix:

```
> matrix(0, 2, 3)           # Create an all-zeros matrix
      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> matrix(NA, 2, 3)          # Create a matrix populated with NA
      [,1] [,2] [,3]
[1,] NA NA NA
[2,] NA NA NA
```



```
[1,] NA NA NA
[2,] NA NA NA
```

You can create a matrix with a one-liner, of course, but it becomes difficult to read:

```
> mat <- matrix(c(1.1, 1.2, 1.3, 2.1, 2.2, 2.3), 2, 3)
```

A common idiom in R is typing the data itself in a rectangular shape that reveals the matrix structure:

```
> theData <- c(1.1, 1.2, 1.3,
+             2.1, 2.2, 2.3)
> mat <- matrix(theData, 2, 3, byrow=TRUE)
```

Setting `byrow=TRUE` tells `matrix` that the data is row-by-row and not column-by-column (which is the default). In condensed form, that becomes:

```
> mat <- matrix(c(1.1, 1.2, 1.3,
+               2.1, 2.2, 2.3),
+               2, 3, byrow=TRUE)
```

Expressed this way, the reader quickly sees the two rows and three columns of data.

There is a quick-and-dirty way to turn a vector into a matrix: just assign dimensions to the vector. This was discussed in the “[Introduction](#)”. The following example creates a vanilla vector and then shapes it into a 2×3 matrix:

```
> v <- c(1.1, 1.2, 1.3, 2.1, 2.2, 2.3)
> dim(v) <- c(2,3)
> v
      [,1] [,2] [,3]
[1,]  1.1  1.3  2.2
[2,]  1.2  2.1  2.3
```

Personally, I find this more opaque than using `matrix`, especially since there is no `byrow` option here.

See Also

See [Recipe 5.3](#).

5.15 Performing Matrix Operations

Problem

You want to perform matrix operations such as transpose, matrix inversion, matrix multiplication, or constructing an identity matrix.

Solution

`t(A)`

Matrix transposition of A

`solve(A)`
Matrix inverse of A

`A %*% B`
Matrix multiplication of A and B

`diag(n)`
An n -by- n diagonal (identity) matrix

Discussion

Recall that `A*B` is element-wise multiplication whereas `A %*% B` is matrix multiplication. All these functions return a matrix. Their arguments can be either matrices or data frames. If they are data frames then R will first convert them to matrices (although this is useful only if the data frame contains exclusively numeric values).

5.16 Giving Descriptive Names to the Rows and Columns of a Matrix

Problem

You want to assign descriptive names to the rows or columns of a matrix.

Solution

Every matrix has a `rownames` attribute and a `colnames` attribute. Assign a vector of character strings to the appropriate attribute:

```
> rownames(mat) <- c("rowname1", "rowname2", ..., "rownamem")
> colnames(mat) <- c("colname1", "colname2", ..., "colnamen")
```

Discussion

R lets you assign names to the rows and columns of a matrix, which is useful for printing the matrix. R will display the names if they are defined, enhancing the readability of your output. Consider this matrix of correlations between the prices of IBM, Microsoft, and Google stock:

```
> print(tech.corr)
      [,1] [,2] [,3]
[1,] 1.000 0.556 0.390
[2,] 0.556 1.000 0.444
[3,] 0.390 0.444 1.000
```

In this form, the matrix output's interpretation is not self-evident. Yet if we define names for the rows and columns, then R will annotate the matrix output with the names:

```

> colnames(tech.corr) <- c("IBM","MSFT","GOOG")
> rownames(tech.corr) <- c("IBM","MSFT","GOOG")
> print(tech.corr)
      IBM  MSFT  GOOG
IBM   1.000 0.556 0.390
MSFT  0.556 1.000 0.444
GOOG  0.390 0.444 1.000

```

Now the reader knows at a glance which rows and columns apply to which stocks.

Another advantage of naming rows and columns is that you can refer to matrix elements by those names:

```

> tech.corr["IBM","GOOG"]    # What is the correlation between IBM and GOOG?
[1] 0.39

```

5.17 Selecting One Row or Column from a Matrix

Problem

You want to select a single row or a single column from a matrix.

Solution

The solution depends on what you want. If you want the result to be a simple vector, just use normal indexing:

```

> vec <- mat[1,]    # First row
> vec <- mat[,3]    # Third column

```

If you want the result to be a one-row matrix or a one-column matrix, then include the `drop=FALSE` argument:

```

> row <- mat[1,,drop=FALSE]    # First row in a one-row matrix
> col <- mat[,3,drop=FALSE]    # Third column in a one-column matrix

```

Discussion

Normally, when you select one row or column from a matrix, R strips off the dimensions. The result is a dimensionless vector:

```

> mat[1,]
[1] 1 4 7 10
> mat[,3]
[1] 7 8 9

```

When you include the `drop=FALSE` argument, however, R retains the dimensions. In that case, selecting a row returns a row vector (a $1 \times n$ matrix):

```

> mat[1,,drop=FALSE]
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10

```

Likewise, selecting a column with `drop=FALSE` returns a column vector (an $n \times 1$ matrix):

```
> mat[,3,drop=FALSE]
      [,1]
[1,]     7
[2,]     8
[3,]     9
```

5.18 Initializing a Data Frame from Column Data

Problem

Your data is organized by columns, and you want to assemble it into a data frame.

Solution

If your data is captured in several vectors and/or factors, use the `data.frame` function to assemble them into a data frame:

```
> dfrm <- data.frame(v1, v2, v3, f1, f2)
```

If your data is captured in a *list* that contains vectors and/or factors, use instead `as.data.frame`:

```
> dfrm <- as.data.frame(list.of.vectors)
```

Discussion

A data frame is a collection of columns, each of which corresponds to an observed variable (in the statistical sense, not the programming sense). If your data is already organized into columns, then it's easy to build a data frame.

The `data.frame` function can construct a data frame from vectors, where each vector is one observed variable. Suppose you have two numeric predictor variables, one categorical predictor variable, and one response variable. The `data.frame` function can create a data frame from your vectors:

```
> dfrm <- data.frame(pred1, pred2, pred3, resp)
> dfrm
      pred1      pred2 pred3      resp
1 -2.7528917 -1.40784130 AM 12.57715
2 -0.3626909  0.31286963 AM 21.02418
3 -1.0416039 -0.69685664 PM 18.94694
4  1.2666820 -1.27511434 PM 18.98153
5  0.7806372 -0.27292745 AM 19.59455
6 -1.0832624  0.73383339 AM 20.71605
7 -2.0883305  0.96816822 PM 22.70062
8 -0.7063653 -0.84476203 PM 18.40691
9 -0.8394022  0.31530793 PM 21.00930
10 -0.4966884 -0.08030948 AM 19.31253
```

Notice that `data.frame` takes the column names from your program variables. You can override that default by supplying explicit column names:

```
> dfrm <- data.frame(p1=pred1, p2=pred2, p3=pred3, r=resp)
> dfrm
      p1      p2 p3      r
1 -2.7528917 -1.40784130 AM 12.57715
2 -0.3626909  0.31286963 AM 21.02418
3 -1.0416039 -0.69685664 PM 18.94694
.
. (etc.)
.
```

Alternatively, your data may be organized into vectors but those vectors are held in a list, not individual program variables, like this:

```
> lst <- list(p1=pred1, p2=pred2, p3=pred3, r=resp)
```

No problem. Use the `as.data.frame` function to create a data frame from the list of vectors:

```
> as.data.frame(lst)
      p1      p2 p3      r
1 -2.7528917 -1.40784130 AM 12.57715
2 -0.3626909  0.31286963 AM 21.02418
3 -1.0416039 -0.69685664 PM 18.94694
.
. (etc.)
.
```

5.19 Initializing a Data Frame from Row Data

Problem

Your data is organized by rows, and you want to assemble it into a data frame.

Solution

Store each row in a one-row data frame. Store the one-row data frames in a list. Use `rbind` and `do.call` to bind the rows into one, large data frame:

```
> dfrm <- do.call(rbind, obs)
```

Here, `obs` is a list of one-row data frames.

Discussion

Data often arrives as a collection of observations. Each *observation* is a record or tuple that contains several values, one for each observed variable. The lines of a flat file are usually like that: each line is one record, each record contains several columns, and each column is a different variable (see [Recipe 4.12](#)). Such data is organized by

observation, not by *variable*. In other words, you are given rows one at a time rather than columns one at a time.

Each such row might be stored in several ways. One obvious way is as a vector. If you have purely numerical data, use a vector.

However, many datasets are a mixture of numeric, character, and categorical data, in which case a vector won't work. I recommend storing each such heterogeneous row in a one-row data frame. (You could store each row in a list, but this recipe gets a little more complicated.)

For concreteness, let's assume that you have ten rows with four variables per observation: `pred1`, `pred2`, `pred3`, and `resp`. Each row is stored in a one-row data frame, so you have ten such data frames. Those data frames are stored in a list called `obs`. The first element of `obs` might look like this:

```
> obs[[1]]
  pred1 pred2 pred3  resp
1 -1.197  0.36   AM 18.701
```

This recipe works also if your observations are stored in vectors rather than one-row data frames.

We need to bind together those rows into a data frame. That's what the `rbind` function does. It binds its arguments in such a way that each argument becomes one row in the result. If we `rbind` the first two observations, for example, we get a two-row data frame:

```
> rbind(obs[[1]], obs[[2]])
  pred1 pred2 pred3  resp
1 -1.197  0.36   AM 18.701
2 -0.952  1.23   PM 25.709
```

We want to bind together every observation, not just the first two, so we tap into the vector processing of R. The `do.call` function will expand `obs` into one, long argument list and call `rbind` with that long argument list:

```
> do.call(rbind,obs)
  pred1 pred2 pred3  resp
1 -1.197  0.360   AM 18.701
2 -0.952  1.230   PM 25.709
3  0.279  0.423   PM 21.572
4 -1.445 -1.846   AM 14.392
5  0.822 -0.246   AM 19.841
6  1.247  1.254   PM 25.637
7 -0.394  1.563   AM 24.585
8 -1.248 -1.264   PM 16.770
9 -0.652 -2.344   PM 14.915
10 -1.171 -0.776   PM 17.948
```

The result is a data frame built from our rows of data.

Sometimes, for reasons beyond your control, the rows of your data are stored in lists rather than one-row data frames. You may be dealing with rows returned by a database package, for example. In that case, `obs` will be a list of lists, not a list of data frames.

We first transform the rows into data frames using the `Map` function and then apply this recipe:

```
> dfrm <- do.call(rbind,Map(as.data.frame,obs))
```

See Also

See [Recipe 5.18](#) if your data is organized by columns, not rows; see [Recipe 12.18](#) to learn more about `do.call`.

5.20 Appending Rows to a Data Frame

Problem

You want to append one or more new rows to a data frame.

Solution

Create a second, temporary data frame containing the new rows. Then use the `rbind` function to append the temporary data frame to the original data frame.

Discussion

Suppose we want to append a new row to our data frame of Chicago-area cities. First, we create a one-row data frame with the new data:

```
> newRow <- data.frame(city="West Dundee", county="Kane", state="IL", pop=5428)
```

Next, we use the `rbind` function to append that one-row data frame to our existing data frame:

```
> suburbs <- rbind(suburbs, newRow)
> suburbs
```

	city	county	state	pop
1	Chicago	Cook	IL	2853114
2	Kenosha	Kenosha	WI	90352
3	Aurora	Kane	IL	171782
4	Elgin	Kane	IL	94487
5	Gary	Lake(IN)	IN	102746
6	Joliet	Kendall	IL	106221
7	Naperville	DuPage	IL	147779
8	Arlington Heights	Cook	IL	76031
9	Bolingbrook	Will	IL	70834
10	Cicero	Cook	IL	72616
11	Evanston	Cook	IL	74239
12	Hammond	Lake(IN)	IN	83048
13	Palatine	Cook	IL	67232
14	Schaumburg	Cook	IL	75386
15	Skokie	Cook	IL	63348
16	Waukegan	Lake(IL)	IL	91452
17	West Dundee	Kane	IL	5428

The `rbind` function tells R that we are appending a new row to `suburbs`, not a new column. It may be obvious to you that `newRow` is a row and not a column, but it is not obvious to R. (Use the `cbind` function to append a column.)

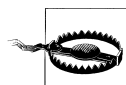
One word of caution. The new row must use the same column names as the data frame. Otherwise, `rbind` will fail.

We can combine these two steps into one, of course:

```
> suburbs <- rbind(suburbs,
+                  data.frame(city="West Dundee", county="Kane", state="IL", pop=5428))
```

We can even extend this technique to multiple new rows because `rbind` allows multiple arguments:

```
> suburbs <- rbind(suburbs,
+                  data.frame(city="West Dundee", county="Kane", state="IL", pop=5428),
+                  data.frame(city="East Dundee", county="Kane", state="IL", pop=2955))
```



Do not use this recipe to append many rows to a large data frame. That would force R to reallocate a large data structure repeatedly, which is a very slow process. Build your data frame using more efficient means, such as those in Recipes [5.19](#) or [5.21](#).

5.21 Preallocating a Data Frame

Problem

You are building a data frame, row by row. You want to preallocate the space instead of appending rows incrementally.

Solution

Create a data frame from generic vectors and factors using the functions `numeric(n)`, `character(n)`, and `factor(n)`:

```
> dfm <- data.frame(colname1=numeric(n), colname2=character(n), ... etc. ... )
```

Here, *n* is the number of rows needed for the data frame.

Discussion

Theoretically, you can build a data frame by appending new rows, one by one. That's OK for small data frames, but building a large data frame in that way can be tortuous. The memory manager in R works poorly when one new row is repeatedly appended to a large data structure. Hence your R code will run very slowly.

One solution is to preallocate the data frame—assuming you know the required number of rows. By preallocating the data frame once and for all, you sidestep problems with the memory manager.

Suppose you want to create a data frame with 1,000,000 rows and three columns: two numeric and one character. Use the `numeric` and `character` functions to preallocate the columns; then join them together using `data.frame`:

```
> N <- 1000000
> dfm <- data.frame(dosage=numeric(N), lab=character(N), response=numeric(N))
```

Now you have a data frame with the correct dimensions, $1,000,000 \times 3$, waiting to receive its contents.

Data frames can contain factors, but preallocating a factor is a little trickier. You can't simply call `factor(n)`. You need to specify the factor's levels because you are creating it. Continuing our example, suppose you want the `lab` column to be a factor, not a character string, and that the possible levels are NJ, IL, and CA. Include the levels in the column specification, like this:

```
> N <- 1000000
> dfm <- data.frame(dosage=numeric(N),
+                  lab=factor(N, levels=c("NJ", "IL", "CA")),
+                  response=numeric(N) )
```

5.22 Selecting Data Frame Columns by Position

Problem

You want to select columns from a data frame according to their position.

Solution

To select a single column, use this list operator:

```
dfm[[n]]
```

Returns *one column*—specifically, the n th column of `dfm`.

To select one or more columns and package them in a data frame, use the following sublist expressions:

```
dfm[n]
```

Returns a *data frame* consisting solely of the n th column of `dfm`.

```
dfm[c(n1, n2, ..., nk)]
```

Returns a *data frame* built from the columns in positions n_1, n_2, \dots, n_k of `dfm`.

You can use matrix-style subscripting to select one or more columns:

```
dfm[, n]
```

Returns the n th column (assuming that n contains exactly one value).

```
dfm[, c(n1, n2, ..., nk)]
```

Returns a *data frame* built from the columns in positions n_1, n_2, \dots, n_k .

Note that the matrix-style subscripting can return two different data types (either column or data frame) depending upon whether you select one column or multiple columns.

Discussion

There are a bewildering number of ways to select columns from a data frame. The choices can be confusing until you understand the logic behind the alternatives. As you read this explanation, notice how a slight change in syntax—a comma here, a double-bracket there—changes the meaning of the expression.

Let's play with the population data for the 16 largest cities in the Chicago metropolitan area:

```
> suburbs
  city      county state    pop
1  Chicago      Cook   IL 2853114
2  Kenosha    Kenosha  WI   90352
3  Aurora      Kane   IL  171782
4  Elgin       Kane   IL   94487
5  Gary      Lake(IN)  IN  102746
6  Joliet     Kendall IL  106221
7  Naperville DuPage  IL  147779
8  Arlington Heights Cook IL   76031
9  Bolingbrook Will  IL   70834
10 Cicero      Cook   IL   72616
11 Evanston    Cook   IL   74239
12 Hammond    Lake(IN) IN   83048
13 Palatine    Cook   IL   67232
14 Schaumburg  Cook   IL   75386
15 Skokie      Cook   IL   63348
16 Waukegan   Lake(IL) IL   91452
```

Use simple list notation to select exactly one column, such as the first column:

```
> suburbs[[1]]
[1] "Chicago"      "Kenosha"      "Aurora"      "Elgin"
[5] "Gary"         "Joliet"       "Naperville"  "Arlington Heights"
[9] "Bolingbrook"  "Cicero"       "Evanston"    "Hammond"
[13] "Palatine"     "Schaumburg"   "Skokie"     "Waukegan"
```

The first column of `suburbs` is a vector, so that's what `suburbs[[1]]` returns: a vector. If the first column were a factor, we'd get a factor.

The result differs when you use the single-bracket notation, as in `suburbs[1]` or `suburbs[c(1,3)]`. You still get the requested columns, but R wraps them in a data frame. This example returns the first column wrapped in a data frame:

```
> suburbs[1]
  city
1  Chicago
2  Kenosha
3  Aurora
4  Elgin
```

```

5           Gary
6         Joliet
7       Naperville
8 Arlington Heights
9     Bolingbrook
10      Cicero
11     Evanston
12      Hammond
13      Palatine
14    Schaumburg
15      Skokie
16    Waukegan

```

The next example returns the first and third columns wrapped in a data frame:

```

> suburbs[c(1,3)]
  city      pop
1  Chicago 2853114
2  Kenosha  90352
3   Aurora 171782
4    Elgin  94487
5     Gary 102746
6    Joliet 106221
7  Naperville 147779
8 Arlington Heights 76031
9  Bolingbrook 70834
10    Cicero 72616
11  Evanston 74239
12  Hammond 83048
13  Palatine 67232
14  Schaumburg 75386
15  Skokie 63348
16  Waukegan 91452

```

A major source of confusion is that `suburbs[[1]]` and `suburbs[1]` look similar but produce very different results:

```
suburbs[[1]]
```

This returns one column.

```
suburbs[1]
```

This returns a data frame, and the data frame contains exactly one column. This is a special case of `dfrm[c(n_1, n_2, \dots, n_k)]`. We don't need the `c(...)` construct because there is only one n .

The point here is that “one column” is different from “a data frame that contains one column.” The first expression returns a column, so it's a vector or a factor. The second expression returns a data frame, which is different.

R lets you use matrix notation to select columns, as shown in the Solution. But an odd quirk can bite you: you might get a column or you might get a data frame, depending upon many subscripts you use. In the simple case of one index you get a column, like this:

```
> suburbs[,1]
[1] "Chicago"      "Kenosha"      "Aurora"       "Elgin"
[5] "Gary"         "Joliet"       "Naperville"   "Arlington Heights"
[9] "Bolingbrook"  "Cicero"       "Evanston"     "Hammond"
[13] "Palatine"     "Schaumburg"   "Skokie"       "Waukegan"
```

But using the same matrix-style syntax with multiple indexes returns a data frame:

```
> suburbs[,c(1,4)]
      city      pop
1   Chicago 2853114
2   Kenosha  90352
3    Aurora 171782
4    Elgin  94487
5    Gary 102746
6    Joliet 106221
7  Naperville 147779
8 Arlington Heights 76031
9  Bolingbrook  70834
10    Cicero  72616
11   Evanston  74239
12    Hammond  83048
13   Palatine  67232
14  Schaumburg  75386
15    Skokie  63348
16   Waukegan  91452
```

This creates a problem. Suppose you see this expression in some old R script:

```
dfrm[,vec]
```

Quick, does that return a column or a data frame? Well, it depends. If `vec` contains one value then you get a column; otherwise, you get a data frame. You cannot tell from the syntax alone.

To avoid this problem, you can include `drop=FALSE` in the subscripts; this forces R to return a data frame:

```
dfrm[,vec,drop=FALSE]
```

Now there is no ambiguity about the returned data structure. It's a data frame.

When all is said and done, using matrix notation to select columns from data frames is not the best procedure. I recommend that you instead use the list operators described previously. They just seem clearer.

See Also

See [Recipe 5.17](#) for more about using `drop=FALSE`.

5.23 Selecting Data Frame Columns by Name

Problem

You want to select columns from a data frame according to their name.

Solution

To select a single column, use one of these list expressions:

```
dfrm[["name"]]
```

Returns *one column*, the column called *name*.

```
dfrm$name
```

Same as previous, just different syntax.

To select one or more columns and package them in a data frame, use these list expressions:

```
dfrm["name"]
```

Selects one column and packages it inside a data frame object.

```
dfrm[c("name1", "name2", ..., "namek")]
```

Selects several columns and packages them in a data frame.

You can use matrix-style subscripting to select one or more columns:

```
dfrm[, "name"]
```

Returns the named column.

```
dfrm[, c("name1", "name2", ..., "namek")]
```

Selects several columns and packages in a data frame.

Once again, the matrix-style subscripting can return two different data types (column or data frame) depending upon whether you select one column or multiple columns.

Discussion

All columns in a data frame must have names. If you know the name, it's usually more convenient and readable to select by name, not by position.

The solutions just described are similar to those for [Recipe 5.22](#), where we selected columns by position. The only difference is that here we use column names instead of column numbers. All the observations made in [Recipe 5.22](#) apply here:

- `dfrm[["name"]]` returns one column, not a data frame.
- `dfrm[c("name1", "name2", ..., "namek")]` returns a data frame, not a column.
- `dfrm["name"]` is a special case of the previous expression and so returns a data frame, not a column.

- The matrix-style subscripting can return either a column or a data frame, so be careful how many names you supply. See [Recipe 5.22](#) for a discussion of this “gotcha” and using `drop=FALSE`.

There is one new addition:

```
dfrm$name
```

This is identical in effect to `dfrm[["name"]]`, but it’s easier to type and to read.

See Also

See [Recipe 5.22](#) to understand these ways to select columns.

5.24 Selecting Rows and Columns More Easily

Problem

You want an easier way to select rows and columns from a data frame or matrix.

Solution

Use the `subset` function. The `select` argument is a column name, or a vector of column names, to be selected:

```
> subset(dfrm, select=colname)
> subset(dfrm, select=c(colname1, ..., colnameN))
```

Note that you do *not* quote the column names.

The `subset` argument is a logical expression that selects rows. Inside the expression, you can refer to the column names as part of the logical expression. In this example, `response` is a column in the data frame, and we are selecting rows with a positive response:

```
> subset(dfrm, subset=(response > 0))
```

`subset` is most useful when you combine the `select` and `subset` arguments:

```
> subset(dfrm, select=c(predictor,response), subset=(response > 0))
```

Discussion

Indexing is the “official” way to select rows and columns from a data frame, as described in [Recipes 5.22](#) and [5.23](#). However, indexing is cumbersome when the index expressions become complicated.

The `subset` function provides a more convenient and readable way to select rows and columns. It’s beauty is that you can refer to the columns of the data frame right inside the expressions for selecting columns and rows.

Here are some examples using the `Cars93` dataset in the `MASS` package. Recall that the dataset includes columns for `Manufacturer`, `Model`, `MPG.city`, `MPG.highway`, `Min.Price`, and `Max.Price`:

Select the model name for cars that can exceed 30 miles per gallon (MPG) in the city

```
> subset(Cars93, select=Model, subset=(MPG.city > 30))
      Model
31 Festiva
39  Metro
42  Civic
.
. (etc.)
.
```

Select the model name and price range for four-cylinder cars made in the United States

```
> subset(Cars93, select=c(Model,Min.Price,Max.Price),
+       subset=(Cylinders == 4 & Origin == "USA"))
      Model Min.Price Max.Price
6    Century      14.2      17.3
12   Cavalier       8.5      18.3
13    Corsica      11.4      11.4
.
. (etc.)
.
```

Select the manufacturer's name and the model name for all cars whose highway MPG value is above the median

```
> subset(Cars93, select=c(Manufacturer,Model),
+       subset=c(MPG.highway > median(MPG.highway)))
      Manufacturer      Model
1         Acura      Integra
5          BMW      535i
6         Buick    Century
.
. (etc.)
.
```

The `subset` function is actually more powerful than this recipe implies. It can select from lists and vectors, too. See the help page for details.

5.25 Changing the Names of Data Frame Columns

Problem

You converted a matrix or list into a data frame. R gave names to the columns, but the names are at best uninformative and at worst bizarre.

Solution

Data frames have a `colnames` attribute that is a vector of column names. You can update individual names or the entire vector:

```
> colnames(dfrm) <- newnames      # newnames is a vector of character strings
```

Discussion

The columns of data frames must have names. If you convert a vanilla matrix into a data frame, R will synthesize names that are reasonable but boring—for example, `V1`, `V2`, `V3`, and so forth:

```
> mat
      [,1] [,2] [,3]
[1,] -0.818 -0.667 -0.494
[2,] -0.819 -0.946 -0.205
[3,]  0.385  1.531 -0.611
[4,] -2.155 -0.535 -0.316
> as.data.frame(mat)
      V1      V2      V3
1 -0.818 -0.667 -0.494
2 -0.819 -0.946 -0.205
3  0.385  1.531 -0.611
4 -2.155 -0.535 -0.316
```

If the matrix had column names defined, R would have used those names instead of synthesizing new ones.

However, converting a list into a data frame produces some strange synthetic names:

```
> lst
[[1]]
[1] -0.284 -1.114 -1.097 -0.873

[[2]]
[1] -1.673  0.929  0.306  0.778

[[3]]
[1]  0.323  0.368  0.067 -0.080

> as.data.frame(lst)
      c..0.284...1.114...1.097...0.873. c..1.673..0.929..0.306..0.778.
1                                -0.284                                -1.673
2                                -1.114                                 0.929
3                                -1.097                                 0.306
4                                -0.873                                 0.778
      c.0.323..0.368..0.067...0.08.
1                                0.323
2                                0.368
3                                0.067
4                                -0.080
```

Again, if the list elements had names then R would have used them.

Fortunately, you can overwrite the synthetic names with names of your own by setting the `colnames` attribute:

```
> dfrm <- as.data.frame(1st)
> colnames(dfrm) <- c("before", "treatment", "after")
> dfrm
  before treatment after
1 -0.284    -1.673  0.323
2 -1.114     0.929  0.368
3 -1.097     0.306  0.067
4 -0.873     0.778 -0.080
```

See Also

See [Recipe 5.33](#).

5.26 Editing a Data Frame

Problem

Data in a data frame are incorrect or missing. You want a convenient way to edit the data frame contents.

Solution

R includes a data editor that displays your data frame in a spreadsheet-like window. Invoke the editor using the `edit` function:

```
> temp <- edit(dfrm)
> dfrm <- temp          # Overwrite only if you're happy with the changes!
```

Make your changes and then close the editor window. The `edit` function will return the updated data, which here are assigned to `temp`. If you are happy with the changes, overwrite your data frame with the results.

If you are feeling brave, the `fix` function invokes the editor and overwrites your variable with the result. There is no “undo”, however:

```
> fix(dfrm)
```

Discussion

[Figure 5-1](#) is a screenshot of the data editor on Windows during the editing of data from [Recipe 5.22](#).

As of this writing, the editor is quite primitive. It does not include the common features of a modern editor—not even an “undo” command, for instance. I cannot recommend the data editor for regular use, but it’s OK for emergency touch-ups.

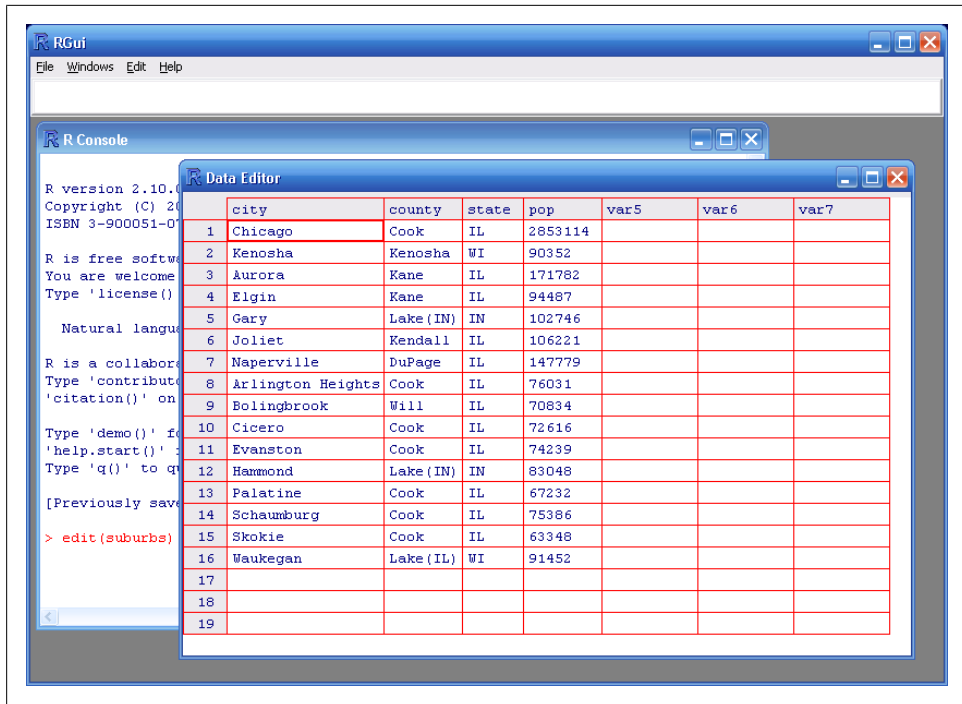


Figure 5-1. Editing a data frame

Since there is no undo, take note of the solution that assigns the edited result to a temporary, intermediate variable. If you mess up your data, you can just delete the temporary variable without affecting the original data.

See Also

Several of the add-on GUI frontends provide data editors that are better than the native editor.

5.27 Removing NAs from a Data Frame

Problem

Your data frame contains NA values, which is creating problems for you.

Solution

Use `na.omit` to remove rows that contain any NA values.

```
> clean <- na.omit(dfm)
```

Discussion

I frequently stumble upon situations where just a few NA values in my data frame cause everything to fall apart. One solution is simply to remove all rows that contain NAs. That's what `na.omit` does.

Here we can see `cumsum` fail because the input contains NA values:

```
> dfm
      x      y
1 -0.9714511 -0.4578746
2      NA    3.1663282
3  0.3367627      NA
4  1.7520504  0.7406335
5  0.4918786  1.4543427
> cumsum(dfm)
      x      y
1 -0.971451 -0.4578746
2      NA    2.7084536
3      NA      NA
4      NA      NA
5      NA      NA
```

If we remove the NA values, `cumsum` can complete its summations:

```
> cumsum(na.omit(dfm))
      x      y
1 -0.9714511 -0.4578746
4  0.7805993  0.2827589
5  1.2724779  1.7371016
```

This recipe works for vectors and matrices, too, but not for lists.

Will You Still Have Enough Data?

The obvious danger here is that simply dropping observations from your data could render the results computationally or statistically meaningless. Make sure that omitting data makes sense in your context. Remember that `na.omit` will remove entire rows, not just the NA values, which could eliminate a lot of useful information.

5.28 Excluding Columns by Name

Problem

You want to exclude a column from a data frame using its name.

Solution

Use the `subset` function with a negated argument for the `select` parameter:

```
> subset(dfm, select = -badboy)      # All columns except badboy
```

Discussion

We can exclude a column by position (e.g., `dfrm[-1]`), but how do we exclude a column by name? The `subset` function can exclude columns from a data frame. The `select` parameter is a normally a list of columns to include, but prefixing a minus sign (-) to the name causes the column to be excluded instead.

I often encounter this problem when calculating the correlation matrix of a data frame and I want to exclude nondata columns such as labels:

```
> cor(patient.data)
      patient.id      pre      dosage      post
patient.id 1.00000000 0.02286906 0.3643084 -0.13798149
pre         0.02286906 1.00000000 0.2270821 -0.03269263
dosage      0.36430837 0.22708208 1.00000000 -0.42006280
post        -0.13798149 -0.03269263 -0.4200628 1.00000000
```

This correlation matrix includes the meaningless “correlation” between patient ID and other variables, which is annoying. We can exclude the patient ID column to clean up the output:

```
> cor(subset(patient.data, select = -patient.id))
      pre      dosage      post
pre     1.00000000 0.2270821 -0.03269264
dosage  0.22708207 1.00000000 -0.42006280
post    -0.03269264 -0.4200628 1.00000000
```

We can exclude multiple columns by giving a vector of negated names:

```
> cor(subset(patient.data, select = c(-patient.id, -dosage)))
      pre      post
pre     1.00000000 -0.03269264
post    -0.03269264 1.00000000
```

See Also

See [Recipe 5.24](#) for more about the `subset` function.

5.29 Combining Two Data Frames

Problem

You want to combine the contents of two data frames into one data frame.

Solution

To combine the columns of two data frames side by side, use `cbind`:

```
> all.cols <- cbind(dfrm1, dfrm2)
```

To “stack” the rows of two data frames, use `rbind`:

```
> all.rows <- rbind(dfrm1, dfrm2)
```

Discussion

You can combine data frames in one of two ways: either by putting the columns side by side to create a wider data frame; or by “stacking” the rows to create a taller data frame. The `cbind` function will combine data frames side by side, as shown here when combining `stooges` and `birth`:

```
> stooges
  name n.married n.child
1  Moe         1       2
2 Larry         1       2
3  Curly        4       2
> birth
 birth.year birth.place
1     1887 Bensonhurst
2     1902 Philadelphia
3     1903   Brooklyn
> cbind(stooges,birth)
  name n.married n.child birth.year birth.place
1  Moe         1       2     1887 Bensonhurst
2 Larry         1       2     1902 Philadelphia
3  Curly        4       2     1903   Brooklyn
```

You would normally combine columns with the same height (number of rows). Technically speaking, however, `cbind` does not require matching heights. If one data frame is short, it will invoke the Recycling Rule to extend the short columns as necessary ([Recipe 5.3](#)), which may or may not be what you want.

The `rbind` function will “stack” the rows of two data frames, as shown here when combining `stooges` and `guys`:

```
> stooges
  name n.married n.child
1  Moe         1       2
2 Larry         1       2
3  Curly        4       2
> guys
  name n.married n.child
1  Tom         4       2
2  Dick        1       4
3  Harry        1       1
> rbind(stooges,guys)
  name n.married n.child
1  Moe         1       2
2 Larry         1       2
3  Curly        4       2
4  Tom         4       2
5  Dick        1       4
6  Harry        1       1
```

The `rbind` function requires that the data frames have the same width: same number of columns and same column names. The columns need not be in the same *order*, however; `rbind` will sort that out.

Finally, this recipe is slightly more general than the title implies. First, you can combine more than two data frames because both `rbind` and `cbind` accept multiple arguments. Second, you can apply this recipe to other data types because `rbind` and `cbind` work also with vectors, lists, and matrices.

See Also

The `merge` function can combine data frames that are otherwise incompatible owing to missing or different columns. The `reshape2` and `plyr` packages, available on CRAN, include some powerful functions for slicing, dicing, and recombining data frames.

5.30 Merging Data Frames by Common Column

Problem

You have two data frames that share a common column. You want to merge their rows into one data frame by matching on the common column.

Solution

Use the `merge` function to join the data frames into one new data frame based on the common column:

```
> m <- merge(df1, df2, by="name")
```

Here *name* is the name of the column that is common to data frames *df1* and *df2*.

Discussion

Suppose you have two data frames, `born` and `died`, that each contain a column called `name`:

```
> born
  name year.born place.born
1  Moe    1887 Bensonhurst
2 Larry   1902 Philadelphia
3  Curly   1903    Brooklyn
4  Harry   1964      Moscow

> died
  name year.died
1 Curly   1952
2  Moe    1975
3 Larry   1975
```

We can merge them into one data frame by using `name` to combine matched rows:

```
> merge(born, died, by="name")
  name year.born place.born year.died
1 Curly   1903    Brooklyn   1952
2 Larry   1902 Philadelphia   1975
3  Moe    1887 Bensonhurst   1975
```

Notice that `merge` does not require the rows to be sorted or even to occur in the same order. It found the matching rows for Curly even though they occur in different positions. It also discards rows that appear in only one data frame or the other.

In SQL terms, the `merge` function essentially performs a join operation on the two data frames. It has many options for controlling that join operation, all of which are described on the help page for `merge`.

See Also

See [Recipe 5.29](#) for other ways to combine data frames.

5.31 Accessing Data Frame Contents More Easily

Problem

Your data is stored in a data frame. You are getting tired of repeatedly typing the data frame name and want to access the columns more easily.

Solution

For quick, one-off expressions, use the `with` function to expose the column names:

```
> with(dataframe, expr)
```

Inside `expr`, you can refer to the columns of `dataframe` by their names—as if they were simple variables.

For repetitive access, use the `attach` function to insert the data frame into your search list. You can then refer to the data frame columns by name without mentioning the data frame:

```
> attach(dataframe)
```

Use the `detach` function to remove the data frame from your search list.

Discussion

A data frame is a great way to store your data, but accessing individual columns can become tedious. For a data frame called `suburbs` that contains a column called `pop`, here is the naïve way to calculate the z-scores of `pop`:

```
> z <- (suburbs$pop - mean(suburbs$pop)) / sd(suburbs$pop)
```

Call me a whiner, but all that typing gets tedious. The `with` function lets you expose the columns of a data frame as distinct variables. It takes two arguments, a data frame and an expression to be evaluated. Inside the expression, you can refer to the data frame columns by their names:

```
> z <- with(suburbs, (pop - mean(pop)) / sd(pop))
```

That is useful for one-liners. If you will be working repeatedly with columns in your data frame, attach the data frame to your search list and the columns will become available as variables:

```
> attach(suburbs)
```

After the `attach`, the second item in the search list is the `suburbs` data frame:

```
> search()
[1] ".GlobalEnv"      "suburbs"          "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "package:methods"   "Autoloads"
[10] "package:base"
```

Now we can refer to the columns of the data frame as if they were variables:

```
> z <- (pop - mean(pop)) / sd(pop)
```

When you are done, use a `detach` (with no arguments) to remove the second location in the search list:

```
> detach()
> search()
[1] ".GlobalEnv"      "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Observe that `suburbs` is no longer in the search list.

Attaching a data frame has a big quirk: R attaches a *temporary copy* of the data frame, which means that changes to the original data frame are hidden. In this session fragment, notice how changing the data frame does not change our view of the attached data:

```
> attach(suburbs)
> pop
[1] 2853114  90352 171782  94487 102746 106221 147779  76031  70834
[10]  72616  74239  83048  67232  75386  63348  91452
> suburbs$pop <- 0      # Overwrite data frame contents
> pop                # Hey! It seems nothing changed
[1] 2853114  90352 171782  94487 102746 106221 147779  76031  70834
[10]  72616  74239  83048  67232  75386  63348  91452
> suburbs$pop        # Contents of data frame did indeed change
[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Another source of confusion is that assigning values to the exposed names does not work as you might expect. In the following fragment, you might think we are scaling `pop` by 1,000 but we are actually creating a new local variable:

```
> attach(suburbs)
> pop
[1] 2853114  90352 171782  94487 102746 106221 147779  76031  70834
[10]  72616  74239  83048  67232  75386  63348  91452
> pop <- pop / 1000    # Achtung! This is creating a local variable called "pop"
> ls()                # We can see the new variable in our workspace
[1] "pop"      "suburbs"
```



```
> suburbs$pop           # Original data is unchanged
[1] 2853114 90352 171782 94487 102746 106221 147779 76031 70834
[10] 72616 74239 83048 67232 75386 63348 91452
```

5.32 Converting One Atomic Value into Another

Problem

You have a data value which has an atomic data type: character, complex, double, integer, or logical. You want to convert this value into one of the other atomic data types.

Solution

For each atomic data type, there is a function for converting values to that type. The conversion functions for atomic types include:

- `as.character(x)`
- `as.complex(x)`
- `as.numeric(x)` or `as.double(x)`
- `as.integer(x)`
- `as.logical(x)`

Discussion

Converting one atomic type into another is usually pretty simple. If the conversion works, you get what you would expect. If it does not work, you get NA:

```
> as.numeric(" 3.14 ")
[1] 3.14
> as.integer(3.14)
[1] 3
> as.numeric("foo")
[1] NA
Warning message:
NAs introduced by coercion
> as.character(101)
[1] "101"
```

If you have a vector of atomic types, these functions apply themselves to every value. So the preceding examples of converting scalars generalize easily to converting entire vectors:

```
> as.numeric(c("1", "2.718", "7.389", "20.086"))
[1] 1.000 2.718 7.389 20.086
> as.numeric(c("1", "2.718", "7.389", "20.086", "etc."))
[1] 1.000 2.718 7.389 20.086 NA
Warning message:
NAs introduced by coercion
```

```
> as.character(101:105)
[1] "101" "102" "103" "104" "105"
```

When converting logical values into numeric values, R converts FALSE to 0 and TRUE to 1:

```
> as.numeric(FALSE)
[1] 0
> as.numeric(TRUE)
[1] 1
```

This behavior is useful when you are counting occurrences of TRUE in vectors of logical values. If `logvec` is a vector of logical values, then `sum(logvec)` does an implicit conversion from logical to integer and returns the number of TRUES.

5.33 Converting One Structured Data Type into Another

Problem

You want to convert a variable from one structured data type to another—for example, converting a vector into a list or a matrix into a data frame.

Solution

These functions convert their argument into the corresponding structured data type:

- `as.data.frame(x)`
- `as.list(x)`
- `as.matrix(x)`
- `as.vector(x)`

Some of these conversions may surprise you, however. I suggest you review [Table 5-1](#).

Discussion

Converting between structured data types can be tricky. Some conversions behave as you'd expect. If you convert a matrix into a data frame, for instance, the rows and columns of the matrix become the rows and columns of the data frame. No sweat.

Table 5-1. Data conversions

Conversion	How	Notes
Vector→List	<code>as.list(vec)</code>	Don't use <code>list(vec)</code> ; that creates a 1-element list whose only element is a copy of <code>vec</code> .
Vector→Matrix	To create a 1-column matrix: <code>cbind(vec)</code> or <code>as.matrix(vec)</code> To create a 1-row matrix: <code>rbind(vec)</code>	See Recipe 5.14 .

Conversion	How	Notes
	To create an $n \times m$ matrix: <code>matrix(vec, n, m)</code>	
Vector→Data frame	To create a 1-column data frame: <code>as.data.frame(vec)</code> To create a 1-row data frame: <code>as.data.frame(rbind(vec))</code>	
List→Vector	<code>unlist(lst)</code>	Use <code>unlist</code> rather than <code>as.vector</code> ; see Note 1 and Recipe 5.11 .
List→Matrix	To create a 1-column matrix: <code>as.matrix(lst)</code> To create a 1-row matrix: <code>as.matrix(rbind(lst))</code> To create an $n \times m$ matrix: <code>matrix(lst, n, m)</code>	
List→Data frame	If the list elements are columns of data: <code>as.data.frame(lst)</code> If the list elements are rows of data: Recipe 5.19	
Matrix→Vector	<code>as.vector(mat)</code>	Returns all matrix elements in a vector.
Matrix→List	<code>as.list(mat)</code>	Returns all matrix elements in a list.
Matrix→Data frame	<code>as.data.frame(mat)</code>	
Data frame→Vector	To convert a 1-row data frame: <code>dfrm[1,]</code> To convert a 1-column data frame: <code>dfrm[, 1]</code> or <code>dfrm[[1]]</code>	See Note 2.
Data frame→List	<code>as.list(dfrm)</code>	See Note 3.
Data frame→Matrix	<code>as.matrix(dfrm)</code>	See Note 4.

In other cases, the results might surprise you. [Table 5-1](#) summarizes some noteworthy examples. The following Notes are cited in that table:

1. When you convert a list into a vector, the conversion works cleanly if your list contains atomic values that are all of the same mode. Things become complicated if either (a) your list contains mixed modes (e.g., numeric and character), in which case everything is converted to characters; or (b) your list contains other structured data types, such as sublists or data frames—in which case very odd things happen, so don't do that.
2. Converting a data frame into a vector makes sense only if the data frame contains one row or one column. To extract all its elements into one, long vector, use `as.vector(as.matrix(dfrm))`. But even that makes sense only if the data frame is all-numeric or all-character; if not, everything is first converted to character strings.
3. Converting a data frame into a list may seem odd in that a data frame is already a list (i.e., a list of columns). Using `as.list` essentially removes the class (`data.frame`) and thereby exposes the underlying list. That is useful when you want R to treat your data structure as a list—say, for printing.

4. Be careful when converting a data frame into a matrix. If the data frame contains only numeric values then you get a numeric matrix. If it contains only character values, you get a character matrix. But if the data frame is a mix of numbers, characters, and/or factors, then all values are first converted to characters. The result is a matrix of character strings.

Problems with matrices

The matrix conversions detailed here assume that your matrix is homogeneous: all elements have the same mode (e.g, all numeric or all character). A matrix can be heterogeneous, too, when the matrix is built from a list. If so, conversions become messy. For example, when you convert a mixed-mode matrix to a data frame, the data frame's columns are actually lists (to accommodate the mixed data).

See Also

See [Recipe 5.32](#) for converting atomic data types; see the [“Introduction”](#) to this chapter for remarks on problematic conversions.