R Data Objects

FRE6871 & FRE7241, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 30, 2022

Data Objects in R

All data objects in R are *vectors*, or consist of *vectors*.

Single numbers and character strings are vectors of length `"1"`.

Atomic vectors are *homogeneous* objects whose elements are all of the same *mode* (type).

Lists and *data frames* are *recursive* (heterogeneous) objects, whose elements can be vectors of different *mode*.

The functions `is.atomic()` and `is.recursive()` return logical values depending on whether their arguments are *atomic* or *recursive*.

R Data Objects

	Atomic	*Recursive*
1-dim	Vectors	Lists
2-dim	Matrices	Data frames
n-dim	Arrays	NA

```
> # Single numbers are vectors of length 1
> 1
[1] 1
> # Character strings are vectors of length 1
> "a"
[1] "a"
> # Strings without quotes are variable names
> a  # Variable "a" doesn't exist
function (..., .noWS = NULL, .renderHook = NULL)
{
    validateNoWS(.noWS)
    contents <- dots_list(...)
    tag("a", contents, .noWS = .noWS, .renderHook = .renderHook)
}
<bytecode: 0x11122fa38>
<environment: namespace:htmltools>
> # List elements can have different mode
> list(aa=c("a", "b"), bb=1:5)
$aa
[1] "a" "b"

$bb
[1] 1 2 3 4 5
> data.frame(aa=c("a", "b"), bb=1:2)
  aa bb
1  a  1
2  b  2
> is.atomic(data.frame(aa=c("a", "b"), bb=1:2))
[1] FALSE
> is.recursive(data.frame(aa=c("a", "b"), bb=1:2))
[1] TRUE
```

Type, Mode, and Class of Objects

The *type*, *mode*, and *class* are `character` strings representing various object properties.

The *type* of an atomic object represents how it's stored in memory (`logical`, `character`, `integer`, `double`, etc.)

The *mode* of an atomic object is the kind of data it represents (`logical`, `character`, `numeric`, etc.)

The *mode* of an object often coincides with its *type* (except for `integer` and `double` types).

Recursive objects (`listv`, `data frames`) have both *type* and *mode* equal to the recursive type (`list`).

The *class* of an object is given by either an explicit *class* attribute, or is derived from the object *mode* and its `dim` attribute (implicit *class*).

The function `class()` returns the explicit or implicit *class* of an object.

The object *class* is used for method dispatching in the S3 object-oriented programming system in R.

```
> myvar <- "hello"
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "character" "character" "character"
>
> myvar <- 1:5
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "integer"
>
> myvar <- runif(5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "numeric"
>
> myvar <- matrix(1:10, 2, 5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "matrix"  "array"
>
> myvar <- matrix(runif(10), 2, 5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "matrix"  "array"
>
> myvar <- list(aa=c("a", "b"), bb=1:5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "list" "list" "list"
>
> myvar <- data.frame(aa=c("a", "b"), bb=1:2)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "list"       "list"       "data.frame"
```

R Object Attributes

R objects can have different attributes, such as: `namesv`, `dimnames`, `dimensions`, *class*, etc.

The attributes of an object is a named `list` of `symbol=value` pairs.

The function `attributes()` returns the attributes of an object.

The attributes of an R object can be modified using the `"attributes() <-"` assignment.

The function `structure()` adds attributes (specified as `symbol=value` pairs) to an object, and returns it.

A `vector` that is assigned an attribute other than `namesv` is not treated as a `vector`.

The function `is.vector()` returns `TRUE` if its argument is a vector, and returns `FALSE` otherwise.

```
> # A simple vector has no attributes
> attributes(5:10)
NULL
> myvar <- c(pi=pi, euler=exp(1), gamma=-digamma(1))
> # Named vector has "namesv" attribute
> attributes(myvar)
$names
[1] "pi"    "euler" "gamma"
> myvar <- 1:10
> is.vector(myvar)  # Is the object a vector?
[1] TRUE
> attributes(myvar) <- list(my_attr="foo")
> myvar
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"my_attr")
[1] "foo"
> is.vector(myvar)  # Is the object a vector?
[1] FALSE
> myvar <- 0
> attributes(myvar) <- list(class="Date")
> myvar  # "Date" object
[1] "1970-01-01"
> structure(0, class="Date")  # "Date" object
[1] "1970-01-01"
```

Modifying *class* Attributes

Objects without an explicit *class* don't have a *class* attribute, and the function `class()` returns the implicit *class*.

The *class* of an object can be modified using the `"class() <-"` assignment.

An object can have a main *class*, and also an inherited *class* (the *class* attribute can be a vector of strings).

The function `unclass()` removes the explicit class attribute from an object.

```
> myvar <- matrix(runif(10), 2, 5)
> class(myvar)  # Has implicit class
[1] "matrix" "array"
> # But no explicit "class" attribute
> attributes(myvar)
$dim
[1] 2 5
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "matrix"  "array"
> # Assign explicit "class" attribute
> class(myvar) <- "my_class"
> class(myvar)  # Has explicit "class"
[1] "my_class"
> # Has explicit "class" attribute
> attributes(myvar)
$dim
[1] 2 5

$class
[1] "my_class"
> is.matrix(myvar)  # Is the object a matrix?
[1] TRUE
> is.vector(myvar)  # Is the object a vector?
[1] FALSE
> attributes(unclass(myvar))
$dim
[1] 2 5
```

Implicit Class of Objects

If an object has no explicit *class*, then its implicit *class* is derived from its *mode* and `dim` attribute (except for `integer` vectors which have the implicit class `"integer"` derived from their *type*).

If an *atomic* object has a `dim` attribute, then its implicit *class* is `matrix` or `array`.

`Data frames` have no explicit `dim` attribute, but `dim()` returns a value, so they have an implicit `dim` attribute.

```
> # Integer implicit class derived from type
> myvar <- vector(mode="integer", length=10)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "integer"
> # Numeric implicit class derived from mode
> myvar <- vector(mode="numeric", length=10)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "numeric"
> # Adding dim attribute changes implicit class to matrix
> dim(myvar) <- c(5, 2)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "matrix"  "array"
> # Data frames have implicit dim attribute
> myvar <- data.frame(aa=c("a", "b"), bb=1:2)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "list"       "list"       "data.frame"
> attributes(myvar)
$names
[1] "aa" "bb"

$class
[1] "data.frame"

$row.names
[1] 1 2
> dim(myvar)
[1] 2 2
```

Object Coercion

Coercion means changing the *type*, *mode*, or *class* of an object, often without changing the underlying data.

Changing the *mode* of an object can change its *class* as well, but not always.

Objects can be explicitly coerced using the `"as.*"` coercion functions.

Most coercion functions strip the *attributes* from the object.

Implicit coercion occurs when objects with different `modes` are combined into a vector, forcing the elements to have the same *mode*.

Implicit coercion can cause bugs that are difficult to trace.

The rule is that coercion is into larger types (`numeric` objects are coerced into `character` strings).

Coercion can introduce bad data, such as `NA` values.

```
> myvar <- 1:5
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "integer"
> mode(myvar) <- "character"  # Coerce to "character"
> myvar
[1] "1" "2" "3" "4" "5"
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "character" "character" "character"
> # Explicitly coerce to "character"
> myvar <- as.character(1:5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "character" "character" "character"
> matrixv <- matrix(1:10, 2, 5)  # Create matrix
> # Explicitly coerce to "character"
> matrixv <- as.character(matrixv)
> c(typeof(matrixv), mode(matrixv), class(matrixv))
[1] "character" "character" "character"
> # Coercion converted matrix to vector
> c(is.matrix(matrixv), is.vector(matrixv))
[1] FALSE  TRUE
> as.logical(0:3)  # Explicit coercion to "logical"
[1] FALSE  TRUE  TRUE  TRUE
> as.numeric(c(FALSE, TRUE, TRUE, TRUE))
[1] 0 1 1 1
> c(1:3, "a")  # Implicit coercion to "character"
[1] "1" "2" "3" "a"
> # Explicit coercion to "numeric"
> as.numeric(c(1:3, "a"))
[1]  1  2  3 NA
```

Basic R Objects

The quotation marks "" (or '') around a character string tell R that it's a string, not a variable name.

Vectors are the basic building blocks of R objects.

There are no scalars in R, and single values are stored as vectors of length "1".

A character string is also a vector with a single element, with the first element of the vector containing the string of text.

The colon binary operator ':' produces a vector.

The function c() combines objects into a vector.

The "[1]" symbol means the return value is a vector.

The function is.vector() returns TRUE if its argument is a vector, and returns FALSE otherwise.

```
> "Hello World!"  # Type some text
> # hello is a variable name, because it's not in quotes
> hello  # R interretsp "hello" as a variable name
> is.vector(1)  # Single number is a vector
> is.vector("a")  # String is a vector
> 4:8  # Create a vector
> # Create vector using c() combine function
> c(1, 2, 3, 4, 5)
> # Create vector using c() combine function
> c("a", "b", "c")
> # Create vector using c() combine function
> c(1, "b", "c")
```

Character Strings

Character `strings` are sequences of characters (and vectors of length one).

The function `nchar()` returns the length of a string.

Special characters in strings:
`"\t"` for TAB,
`"\n"` for new-line,
"`\\`" for a (single) backslash character

The function `cat()` concatenates strings and echos them to console, without returning any values.

The function `cat()` is useful in user-defined functions.

```
> stringv <- "Some string"
> stringv
[1] "Some string"
> stringv[1]
[1] "Some string"
> stringv[2]
[1] NA
>
> NROW(stringv)  # length of vector
[1] 1
> nchar(stringv)  # length of string
[1] 11
>
> # Concatenate and echo to console
> cat("Hello", "World!")
Hello World!
> cat("Enter\ttab")
Enter tab
> cat("Enter\nnewline")
Enter
newline
> cat("Enter\\backslash")
Enter\backslash
```

Manipulating Strings

The function `paste()` concatenates its arguments into a string, coerces them to characters if needed, and returns the string.

If a vector or list is passed to `paste()`, together with a `collapse` string, then `paste()` concatenates the elements into a string, separated by the `collapse` string.

The function `strsplit()` splits the elements of a character vector.

Splitting on the `"."` character requires surrounding it with brackets: `"[.]"`, or using argument `fixed=TRUE`.

The function `substring()` extracts or replaces substrings in a character string.

The recycling rule extends the length to match the longest object.

```
> stringv1 <- "Hello"  # Define a character string
> stringv2 <- "World!"  # Define a character string
> paste(stringv1, stringv2, sep=" ")  # Concatenate and return value
[1] "Hello World!"
> cat(stringv1, stringv2)  # Concatenate and echo to console
Hello World!
> paste("a", 1:4, sep="-")  # Convert, recycle and concatenate
[1] "a-1" "a-2" "a-3" "a-4"
> paste(c("a1", "a2", "a3"), collapse="+")  # Collapse vector to str
[1] "a1+a2+a3"
> paste(list("a1", "a2", "a3"), collapse="+")
[1] "a1+a2+a3"
> paste("Today is", Sys.time())  # Coerce and concatenate strings
[1] "Today is 2022-10-30 19:55:55"
> paste("Today is", format(Sys.time(), "%B-%d-%Y"))
[1] "Today is October-30-2022"
> strsplit("Hello World", split="r")  # Split string
[[1]]
[1] "Hello Wo" "ld"
> strsplit("Hello.World", split="[.]")  # Split string
[[1]]
[1] "Hello" "World"
> strsplit("Hello.World", split=".", fixed=TRUE)  # Split string
[[1]]
[1] "Hello" "World"
> substring("Hello World", 3, 6)  # Extract characters from 3 to 6
[1] "llo "
```

Regular Expressions in R

R has Regex functions for pattern matching and replacement.

The function gsub() replaces all matches of a pattern in a string.

The function grep() searches for matches of a pattern in a string.

The function glob2rx() converts globbing wildcard patterns into regular expressions.

```
> gsub("is", "XX", "is this gratis?")  # Replace "is" with "XX"
[1] "XX thXX gratXX?"
>
> grep("b", c("abc", "xyz", "cba d", "bbb"))  # Get indexes
[1] 1 3 4
>
> grep("b", c("abc", "xyz", "cba d", "bbb"), value=TRUE)  # Get valu
[1] "abc"   "cba d" "bbb"
>
> glob2rx("abc.*")  # Convert globs into regex
[1] "^abc\\."
> glob2rx("*.doc")
[1] "^.*\\.doc$"
```

Vectors

Vectors are the basic building blocks of R objects.

There are no scalars in R, and single values are stored as vectors of length "1".

The function c() combines values into a vector.

The function is.vector() returns TRUE if its argument is a vector, and returns FALSE otherwise.

The object letters is a constant and a vector,

```
> is.vector(1)  # Single number is a vector
[1] TRUE
> is.vector("a")  # String is a vector
[1] TRUE
> vectorv <- c(8, 6, 5, 7)  # Create vector
> vectorv
[1] 8 6 5 7
> vectorv[2]  # Extract second element
[1] 6
> # Extract all elements, except the second element
> vectorv[-2]
[1] 8 5 7
> # Create Boolean vector
> c(FALSE, TRUE, TRUE)
[1] FALSE  TRUE  TRUE
> # Extract second and third elements
> vectorv[c(FALSE, TRUE, TRUE)]
[1] 6 5
> letters[5:10]  # Vector of letters
[1] "e" "f" "g" "h" "i" "j"
> c("a", letters[5:10])  # Combine two vectors of letters
[1] "a" "e" "f" "g" "h" "i" "j"
```

Creating Vectors

The colon operator (":") provides a simple way of creating a numeric vector.

The function `vector()` returns a vector of the specified *mode*.

The functions `seq()`, `seq_len()`, and `seq_along()` return a sequence (vector) of numbers.

The function `rep()` replicates an object multiple times.

The functions `character()` and `numeric()` return zero-length vectors of the specified *mode* (not to be confused with a `NULL` object).

Zero length vectors are not the same as `NULL` objects.

```
> 0:10  # Vector of integers from 0 to 10
 [1]  0  1  2  3  4  5  6  7  8  9 10
> vector()  # Create empty vector
logical(0)
> vector(mode="numeric", length=10)  # Numeric vector of zeros
 [1] 0 0 0 0 0 0 0 0 0 0
> seq(10)  # Sequence from 1 to 10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(along=(-5:5))  # Instead of 1:NROW(obj)
 [1]  1  2  3  4  5  6  7  8  9 10 11
> seq_along(c("a", "b", "c"))  # Instead of 1:NROW(obj)
[1] 1 2 3
> seq(from=0, to=1, len=11)  # Decimals from 0 to 1.0
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(from=0, to=1, by=0.1)  # Decimals from 0 to 1.0
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(-2,2, len=11)  # 10 numbers from -2 to 2
 [1] -2.0 -1.6 -1.2 -0.8 -0.4  0.0  0.4  0.8  1.2  1.6  2.0
> rep(100, times=5)  # Replicate a number
[1] 100 100 100 100 100
> character(5)  # Create empty character vector
[1] "" "" "" "" ""
> numeric(5)  # Create empty numeric vector
[1] 0 0 0 0 0
> numeric(0)  # Create zero-length vector
numeric(0)
```

Arithmetic and Logical Operations on Vectors

Vectors can be multiplied and squared element by element, as if they were single elements.

When vectors are manipulated as if they were single elements, then R automatically performs a loop over the vector elements, and applies the operation element-wise.

This is a very powerful feature of R called *vectorized arithmetic*.

Vectorized arithmetic avoids writing loops and simplifies notation.

Vectors can be combined together and appended.

```
> 2*4:8  # Multiply a vector
> 2*(4:8)  # Multiply a vector
> 4:8/2  # Divide a vector
> (0:10)/10  # Divide vector - decimals from 0 to 1.0
> vectorv <- c(8, 6, 5, 7)  # Create vector
> vectorv
> # Boolean vector TRUE if element is equal to second one
> vectorv == vectorv[2]
> # Boolean vector TRUE for elements greater than six
> vectorv > 6
> 2*vectorv  # Multiply all elements by 2
> vectorv^2  # Square all elements
> c(11, 5:10)  # Combine two vectors
> c(vectorv, 2.0)  # Append number to vector
```

Naming and Manipulating Vectors

Vector elements can be assigned `namesv` using a list of symbol-value pairs.

The function `names()` returns the `namesv` attribute of an object.

The `namesv` attribute of a vector can be modified by assigning to the `names()` function ("`names() <-`" assignment).

The function `unname()` removes the `namesv` attribute.

The function `structure()` adds attributes (specified as `symbol=value` pairs) to an object, and returns it.

```
> vectorv <- # Create named vector
+   c(pi_const=pi, euler=exp(1), gamma=-digamma(1))
> vectorv
pi_const    euler    gamma
   3.142    2.718    0.577
> names(vectorv)  # Get names of elements
[1] "pi_const" "euler"    "gamma"
> vectorv["euler"]  # Get element named "euler"
euler
 2.72
> names(vectorv) <- c("pie","eulery","gammy")  # Rename elements
> vectorv
   pie eulery  gammy
 3.142  2.718  0.577
> unname(vectorv)  # Remove names attribute
[1] 3.142 2.718 0.577
> letters[5:10]  # Vector of letters
[1] "e" "f" "g" "h" "i" "j"
> c("a", letters[5:10])  # Combine two vectors of letters
[1] "a" "e" "f" "g" "h" "i" "j"
> # Create named vector
> structure(sample(1:5), names=paste0("el", 1:5))
el1 el2 el3 el4 el5
  5   4   1   2   3
```

Subsetting Vectors

Vector elements can be *subset* (indexed, referenced) using the "[]" operator.

Vectors can be *subset* using vectors of:

- positive integers,
- negative integers,
- characters (names),
- Boolean vectors,

Negative integers remove the vector elements.

Subsetting with *zero* returns a zero-length vector.

A named vector can be *subset* using element names.

```
> vectorv  # Named vector
   pie eulery  gammy
 3.142  2.718  0.577
> # Extract second element
> vectorv[2]
eulery
  2.72
> # Extract all elements, except the second element
> vectorv[-2]
  pie gammy
3.142 0.577
> # Extract zero elements - returns zero-length vector
> vectorv[0]
named numeric(0)
> # Extract second and third elements
> vectorv[c(FALSE, TRUE, TRUE)]
eulery  gammy
 2.718  0.577
> # Extract elements using their names
> vectorv["eulery"]
eulery
  2.72
> # Extract elements using their names
> vectorv[c("pie", "gammy")]
  pie gammy
3.142 0.577
> # Subset whole vector
> vectorv[] <- 0
```

Filtering Vectors

Filtering means extracting elements from a vector that satisfy a logical condition.

When logical comparison operators are applied to vectors, they produce `Boolean` vectors.

`Boolean` vectors can then be applied to subset the original vectors, to extract their elements.

The function `which()` returns the indices of the `TRUE` elements of a `Boolean vector` or `array`.

```
> vectorv <- runif(5)
> vectorv
[1] 0.157 0.462 0.123 0.545 0.101
> vectorv > 0.5  # Boolean vector
[1] FALSE FALSE FALSE  TRUE FALSE
> # Boolean vector of elements equal to the second one
> vectorv == vectorv[2]
[1] FALSE  TRUE FALSE FALSE FALSE
> # Extract all elements equal to the second one
> vectorv[vectorv == vectorv[2]]
[1] 0.462
> vectorv < 1  # Boolean vector of elements less than one
[1] TRUE TRUE TRUE TRUE TRUE
> # Extract all elements greater than one
> vectorv[vectorv > 1]
numeric(0)
> vectorv[vectorv > 0.5]  # Filter elements > 0.5
[1] 0.545
> which(vectorv > 0.5)  # Index of elements > 0.5
[1] 4
```

Factors

Factors are similar to vectors, but their elements can only take values from a set of *levels*.

Factors are designed for categorical data which can only take certain values.

The function `factor()` converts a vector into a factor.

Factors have two attributes: *class* (equal to `"factor"`) and `levels` (the allowed values).

Although factors aren't vectors, the data underlying a factor is an integer vector, called an *encoding vector*.

The function `as.numeric()` extracts the encoding vector (indices) of a factor.

The function `as.vector()` coerces a factor to a character vector.

```
> # Create factor vector
> factorv <- factor(c("b", "c", "d", "a", "c", "b"))
> factorv
[1] b c d a c b
Levels: a b c d
> factorv[3]
[1] d
Levels: a b c d
> # Get factor attributes
> attributes(factorv)
$levels
[1] "a" "b" "c" "d"

$class
[1] "factor"
> # Get allowed values
> levels(factorv)
[1] "a" "b" "c" "d"
> # Get encoding vector
> as.numeric(factorv)
[1] 2 3 4 1 3 2
> is.vector(factorv)
[1] FALSE
> # Coerce vector to factor
> as.factor(1:5)
[1] 1 2 3 4 5
Levels: 1 2 3 4 5
> # Coerce factor to character vector
> as.vector(as.factor(1:5))
[1] "1" "2" "3" "4" "5"
```

Tables of Categorical Data

The function `unique()` calculates the unique elements of an object.

The function `levels()` extracts the `levels` attribute of a factor (the allowed values).

A contingency table is a matrix that contains the frequency distribution of variables (factors) contained in a set of data.

The function `table()` calculates the frequency distribution of categorical data.

`sapply()` applies a function to a vector or a list of objects and returns a vector or a list.

```
> # Print factor vector
> factorv
[1] b c d a c b
Levels: a b c d
> # Get unique elements of factorv
> unique(factorv)
[1] b c d a
Levels: a b c d
> # Get levels attribute of factorv
> levels(factorv)
[1] "a" "b" "c" "d"
> # Calculate the factor elements from its levels
> levels(factorv)[as.numeric(factorv)]
[1] "b" "c" "d" "a" "c" "b"
> # Get contingency (frequency) table
> table(factorv)
factorv
a b c d
1 2 2 1
```

Classifying Continuous Numeric Data Into Categories

Numeric data that represents a *magnitude*, *intensity*, or *score* can be classified into categorical data, given a vector of *breakpoints*.

The *breakpoints* create intervals that correspond to different *categories*.

The *categories* combine elements that have a similar numeric *magnitude*.

`findInterval()` returns the indices of the intervals specified by `"vec"` that contain the elements of `"x"`.

If there's an exact match, then `findInterval()` returns the same index as function `match()`.

If there's no exact match, then `findInterval()` finds the element of `"vec"` that is closest to, but not greater than, the element of `"x"`.

If all the elements of `"vec"` are greater than the element of `"x"`, then `findInterval()` returns `zero`.

`args()` displays the formal arguments of a function.

```
> # Display the formal arguments of findInterval
> args(findInterval)
function (x, vec, rightmost.closed = FALSE, all.inside = FALSE,
    left.open = FALSE)
NULL
> # Get index of the element of "vec" that matches 5
> findInterval(x=5, vec=c(3, 5, 7))
[1] 2
> match(5, c(3, 5, 7))
[1] 2
> # No exact match
> findInterval(x=6, vec=c(3, 5, 7))
[1] 2
> match(6, c(3, 5, 7))
[1] NA
> # Indices of "vec" that match elements of "x"
> findInterval(x=1:8, vec=c(3, 5, 7))
[1] 0 0 1 1 2 2 3 3
> # Return only indices of inside intervals
> findInterval(x=1:8, vec=c(3, 5, 7), all.inside=TRUE)
[1] 1 1 1 1 2 2 2 2
> # make rightmost interval inclusive
> findInterval(x=1:8, vec=c(3, 5, 7), rightmost.closed=TRUE)
[1] 0 0 1 1 2 2 2 3
```

Classifying Numeric Data Into Categories Example

Temperature can be categorized into "cold", "warm", "hot", etc.

A named numeric vector of *breakpoints* can be used to convert a temperature into one of the *categories*.

Breakpoints correspond to *categories* of the data.

The first *breakpoint* should correspond to the lowest *category*, and should have a value less than any of the data.

```
> # Named numeric vector of breakpoints
> brea_ks <- c(freezing=0, very_cold=30, cold=50,
+        pleasant=60, warm=80, hot=90)
> brea_ks
 freezing very_cold      cold  pleasant      warm       hot
        0        30        50        60        80        90
> tempe_ratures <- runif(10, min=10, max=100)
> feels_like <- names(
+   brea_ks[findInterval(x=tempe_ratures, vec=brea_ks)])
> names(tempe_ratures) <- feels_like
> tempe_ratures
 pleasant      warm      warm  pleasant very_cold very_cold      wa
     79.8      88.6      85.0      78.8      35.8      38.7      83
very_cold very_cold
     43.0      38.4
```

Converting Numeric Data Into Factors Using cut()

The function `cut()` converts a numeric vector into a vector of factors, representing the intervals to which the numeric values belong.

`cut()` divides the range of values into intervals, based on a vector of breaks.

`cut()` then assigns factors to the numeric values, representing the intervals to which the numeric values belong.

The parameter `breaks` is a numeric vector of break points that divide the range of values into intervals.

The argument `"labels"` is a vector of labels for the intervals.

The argument `"right"` is a Boolean indicating if the intervals should be closed on the right (and open on the left), or vice versa.

`cut()` can produce the same classification as `findInterval()`, but `findInterval()` is faster than `cut()`, because it's a compiled function.

```
> datav <- sample(0:6) + 0.1
> datav
[1] 4.1 2.1 1.1 6.1 3.1 0.1 5.1
> cut(x=datav, breaks=c(2, 4, 6, 8))
[1] (4,6] (2,4] <NA>  (6,8] (2,4] <NA>  (4,6]
Levels: (2,4] (4,6] (6,8]
> rbind(datav, cut(x=datav, breaks=c(2, 4, 6, 8)))
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
datav  4.1  2.1  1.1  6.1  3.1  0.1  5.1
       2.0  1.0   NA  3.0  1.0   NA  2.0
> # cut() replicates findInterval()
> cut(x=1:8, breaks=c(3, 5, 7), labels=1:2,
+     right=FALSE)
[1] <NA> <NA> 1    1    2    2    <NA> <NA>
Levels: 1 2
> findInterval(x=1:8, vec=c(3, 5, 7))
[1] 0 0 1 1 2 2 3 3
> # findInterval() is a compiled function, so it's faster than cut()
> vectorv <- rnorm(1000)
> summary(microbenchmark(
+   find_interval=
+     findInterval(x=vectorv, vec=c(3, 5, 7)),
+   cuut=
+     cut(x=vectorv, breaks=c(3, 5, 7)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
          expr  mean median
1 find_interval  4.84    4.2
2          cuut 67.47   57.7
```

Plotting Histograms of Frequency Data

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

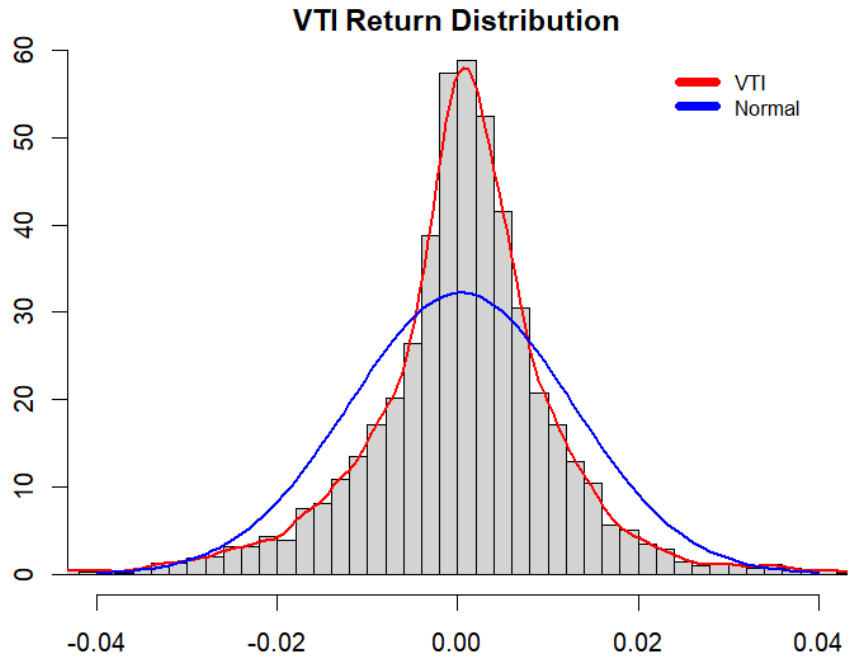
The parameter `breaks` is the number of cells of the histogram.

If the argument `freq` is `TRUE` then the frequencies (counts) are plotted, and if it's `FALSE` then the probability density is plotted (with total area equal to 1).

The function `density()` calculates a kernel estimate of the probability density for a sample of data.

The function `lines()` draws a line through specified points.

```
> # Calculate VTI percentage returns
> retsp <- na.omit(rutils::etfenv$returns$VTI)
> # Plot histogram
> x11(width=6, height=5)
> par(mar=c(1, 1, 1, 1), oma=c(2, 2, 2, 0))
> madv <- mad(retsp)
> histp <- hist(retsp, breaks=100,
+   main="", xlim=c(-5*madv, 5*madv),
+   xlab="", ylab="", freq=FALSE)
```




```
> # Draw kernel density of histogram
> lines(density(retsp), col="red", lwd=2)
> # Add density of normal distribution
> curve(expr=dnorm(x, mean=mean(retsp), sd=sd(retsp)),
+ add=TRUE, type="l", lwd=2, col="blue")
> title(main="VTI Return Distribution", line=0)
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+   leg=c("VTI", "Normal"), bty="n",
+   lwd=6, bg="white", col=c("red", "blue"))
> # Total area under histogram
> sum(diff(histp$breaks) * histp$density)
```

Matrices

The function `matrix()` creates a matrix from a vector, and the matrix dimensions.

By default `matrix()` creates matrices column-wise, unless the argument `byrow=TRUE` is used.

The elements of matrices can be subset (referenced) using the `"[]"` operator.

The functions `nrow()` and `ncol()` return the number of rows and columns of a matrix.

The functions `NROW()` and `NCOL()` also return the number of rows or columns of a matrix, but they can also be applied to vectors, and treat vectors as single column matrices.

```
> matrixv <- matrix(5:10, nrow=2, ncol=3)  # Create a matrix
> matrixv  # By default matrices are constructed column-wise
     [,1] [,2] [,3]
[1,]    5    7    9
[2,]    6    8   10
> # Create a matrix row-wise
> matrix(5:10, nrow=2, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    5    6    7
[2,]    8    9   10
> matrixv[2, 3]  # Extract third element from second row
[1] 10
> matrixv[2, ]  # Extract second row
[1]  6  8 10
> matrixv[, 3]  # Extract third column
[1]  9 10
> matrixv[, c(1,3)]  # Extract first and third column
     [,1] [,2]
[1,]    5    9
[2,]    6   10
> matrixv[, -2]  # Remove second column
     [,1] [,2]
[1,]    5    9
[2,]    6   10
> # Subset whole matrix
> matrixv[] <- 0
> # Get the number of rows or columns
> nrow(vectorv); ncol(vectorv)
NULL
NULL
> NROW(vectorv); NCOL(vectorv)
[1] 1000
[1] 1
> nrow(matrixv); ncol(matrixv)
[1] 2
[1] 3
> NROW(matrixv); NCOL(matrixv)
[1] 2
```

Matrix Attributes

Arrays are vectors with a dimension attribute.

Matrices are two-dimensional arrays.

The dimension attribute of a matrix is an integer vector of length 2 (nrow, ncol).

The `dimnames` attribute is a list, with vector elements containing row and column names.

A named matrix can be subset using row and column names.

```
> attributes(matrixv)  # Get matrix attributes
$dim
[1] 2 3
> dim(matrixv)  # Get dimension attribute
[1] 2 3
> class(matrixv)  # Get class attribute
[1] "matrix" "array"
> rownames(matrixv) <- c("row1", "row2")  # Rownames attribute
> colnames(matrixv) <- c("col1", "col2", "col3")  # Colnames attribu
> matrixv
     col1 col2 col3
row1    0    0    0
row2    0    0    0
> matrixv["row2", "col3"]  # Third element from second row
[1] 0
> names(matrixv)  # Get the names attribute
NULL
> dimnames(matrixv)  # Get dimnames attribute
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"
> attributes(matrixv)  # Get matrix attributes
$dim
[1] 2 3

$dimnames
$dimnames[[1]]
[1] "row1" "row2"

$dimnames[[2]]
[1] "col1" "col2" "col3"
```

Matrix Subsetting

Matrices can be subset in a similar way as Vectors, either by indices (integers), by characters (names), or `Boolean` vectors.

Subsetting a matrix to a single row or column produces a vector, unless the parameter `"drop=FALSE"` is used.

Subsetting with the parameter `"drop=FALSE"` prevents the implicit coercion and preserves the matrix *class*.

This is an example of implicit coercion in `R`, which can cause difficult to trace bugs.

```
> matrixv  # matrix with column names
     col1 col2 col3
row1    0    0    0
row2    0    0    0
> matrixv[1, ]  # Subset rows by index
col1 col2 col3
   0    0    0
> matrixv[, "col1"]  # Subset columns by name
row1 row2
   0    0
> matrixv[, c(TRUE, FALSE, TRUE)]  # Subset columns Boolean vector
     col1 col3
row1    0    0
row2    0    0
> matrixv[1, ]  # Subsetting can produce a vector!
col1 col2 col3
   0    0    0
> class(matrixv); class(matrixv[1, ])
[1] "matrix" "array"
[1] "numeric"
> is.matrix(matrixv[1, ]); is.vector(matrixv[1, ])
[1] FALSE
[1] TRUE
> matrixv[1, , drop=FALSE]  # Drop=FALSE preserves matrix
     col1 col2 col3
row1    0    0    0
> class(matrixv[1, , drop=FALSE])
[1] "matrix" "array"
> is.matrix(matrixv[1, , drop=FALSE]); is.vector(matrixv[1, , drop=F
[1] TRUE
[1] FALSE
```

Lists

Lists are a type of vector that contain elements of different *types*.

Lists are recursive object types, meaning each list element can contain other vectors or lists.

The function `list()` creates a list from a list of vectors.

`list()` creates a named list from a list of symbol-value pairs.

The function `is.list()` returns `TRUE` if its argument is a list, and `FALSE` otherwise.

The function `unlist()` collapses a list with atomic elements into a vector (which can cause type coercion).

```
> # Create a list with two elements
> listv <- list(c("a", "b"), 1:4)
> listv
[[1]]
[1] "a" "b"

[[2]]
[1] 1 2 3 4
> c(typeof(listv), mode(listv), class(listv))
[1] "list" "list" "list"
> # Lists are also vectors
> c(is.vector(listv), is.list(listv))
[1] TRUE TRUE
> NROW(listv)
[1] 2
> # Create named list
> listv <- list(first=c("a", "b"), second=1:4)
> listv
$first
[1] "a" "b"

$second
[1] 1 2 3 4
> names(listv)
[1] "first"  "second"
> unlist(listv)
 first1  first2 second1 second2 second3 second4
    "a"     "b"     "1"     "2"     "3"     "4"
```

Subsetting Lists

Lists can be subset (indexed) using:

- the "[" operator (returns sublist),
- the "[[" operator (returns an element),
- the "$" operator (for named listv only),

Partial name matching allows subsetting with partial name, as long as it can be resolved.

```
> listv[2]  # Extract second element as sublist
$second
[1] 1 2 3 4
> listv[[2]]  # Extract second element
[1] 1 2 3 4
> listv[[2]][3]  # Extract third element of second element
[1] 3
> listv[[c(2, 3)]]  # Third element of second element
[1] 3
> listv$second  # Extract second element
[1] 1 2 3 4
> listv$s  # Extract second element - partial name matching
[1] 1 2 3 4
> listv$second[3]  # Third element of second element
[1] 3
> listv <- list()  # Empty list
> listv$a <- 1
> listv[2] <- 2
> listv
$a
[1] 1

[[2]]
[1] 2
> names(listv)
[1] "a" ""
```

Coercing Vectors Into Lists Using `as.list()`

The function `as.list()` coerces vectors and other objects into lists.

`as.list()` returns a list with the same elements as the vector.

`list()` called on a vector returns a single element equal to the vector.

```
> # Convert vector elements to list elements
> as.list(1:3)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
> # Convert whole vector to single list element
> list(1:3)
[[1]]
[1] 1 2 3
```

Data Frames

Data frames are 2-D objects (like matrices), but their columns can be of different *types*.

Data frames can be thought of as listv of vectors of the same length.

The function `data.frame()` creates a *data frame* from vectors assigned to column names.

```
> dframe <- data.frame(  # Create a data frame
+                 type=c("rose", "daisy", "tulip"),
+                 color=c("red", "white", "yellow"),
+                 price=c(1.5, 0.5, 1.0)
+               )  # end data.frame
> dframe
   type  color price
1  rose    red   1.5
2 daisy  white   0.5
3 tulip yellow   1.0
> dim(dframe)  # Get dimension attribute
[1] 3 3
> colnames(dframe)  # Get the colnames attribute
[1] "type"  "color" "price"
> rownames(dframe)  # Get the rownames attribute
[1] "1" "2" "3"
> class(dframe)  # Get object class
[1] "data.frame"
> typeof(dframe)  # Data frames are listv
[1] "list"
> is.data.frame(dframe)
[1] TRUE
>
> class(dframe$type)  # Get column class
[1] "character"
> class(dframe$price)  # Get column class
[1] "numeric"
```

Subsetting Data Frames

Data frames can be subset in a similar way to listv and matrices.

Depending on how a data frame is subset, the result can be either a data frame or a vector.

Extracting a single column from a data frame produces a vector.

The data frame class attribute can be preserved by using the parameter `"drop=FALSE"`.

Extracting a single row from a data frame produces a data frame.

The function `unlist()` applied to a single row extracted from a data frame coerces it to a vector.

```
> dframe[, 3]  # Extract third column as vector
[1] 1.5 0.5 1.0
> dframe[[3]]  # Extract third column as vector
[1] 1.5 0.5 1.0
> dframe[3]  # Extract third column as data frame
  price
1   1.5
2   0.5
3   1.0
> dframe[, 3, drop=FALSE]  # Extract third column as data frame
  price
1   1.5
2   0.5
3   1.0
> dframe[[3]][2]  # Second element from third column
[1] 0.5
> dframe$price[2]  # Second element from "price" column
[1] 0.5
> is.data.frame(dframe[[3]]); is.vector(dframe[[3]])
[1] FALSE
[1] TRUE
> dframe[2, ]  # Extract second row
   type color price
2 daisy white   0.5
> dframe[2, ][3]  # Third element from second column
  price
2   0.5
> dframe[2, 3]  # Third element from second column
[1] 0.5
> unlist(dframe[2, ])  # Coerce to vector
   type   color   price
"daisy" "white"   "0.5"
> is.data.frame(dframe[2, ]); is.vector(dframe[2, ])
[1] TRUE
[1] FALSE
```

Data Frames and Factors

By default `data.frame()` does not coerce `character` vectors to `factors`, so no need for the option `stringsAsFactors=FALSE`.

The function `options()` sets global *options*, that determine how `R` computes and displays its results.

If the global `option stringsAsFactors=FALSE` is set, then `character` vectors will not be coerced to `factors` in all subsequent `data frame` operations.

The default is `stringsAsFactors=FALSE` since `R` version `4.0`.

```
> dframe <- data.frame(  # Create a data frame
+                 type=c("rose", "daisy", "tulip"),
+                 color=c("red", "white", "yellow"),
+                 price=c(1.5, 0.5, 1.0),
+                 row.names=c("flower1", "flower2", "flower3")
+               )  # end data.frame
> dframe
         type  color price
flower1  rose    red   1.5
flower2 daisy  white   0.5
flower3 tulip yellow   1.0
> class(dframe$type)  # Get column class
[1] "character"
> class(dframe$price)  # Get column class
[1] "numeric"
> # Set option to not coerce character vectors to factors
> options("stringsAsFactors")
$stringsAsFactors
[1] FALSE
> default.stringsAsFactors()
[1] FALSE
> options(stringsAsFactors=FALSE)
```

Exploring Data Frames

The function `str()` displays the structure of an R object.

The functions `head()` and `tail()` display the first and last rows of an R object.

```
> str(dframe)  # Display the object structure
'data.frame': 3 obs. of  3 variables:
 $ type : chr  "rose" "daisy" "tulip"
 $ color: chr  "red" "white" "yellow"
 $ price: num  1.5 0.5 1
> dim(cars)  # The cars data frame has 50 rows
[1] 50  2
> head(cars, n=5)  # Get first five rows
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
> tail(cars, n=5)  # Get last five rows
   speed dist
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85
```

Sorting Vectors

The function `sort()` returns a vector sorted into ascending order.

A permutation is a re-ordering of the elements of a vector.

The permutation index specifies how the elements are re-ordered in a permutation.

The function `order()` calculates the permutation index to sort a given vector into ascending order.

Applying the function `order()` twice: `order(order())`, calculates the permutation index to sort the vector from ascending order into its unsorted (original) order.

So the permutation index produced by: `order(order())` is the reverse of the permutation index produced by: `order()`.

`order()` can take several vectors as input, to break any ties.

Data frames can be sorted on any column.

```
> # Create a named vector of student scores
> scorev <- sample(round(runif(5, min=1, max=10), digits=2))
> names(scorev) <- c("Angie", "Chris", "Suzie", "Matt", "Liz")
> # Sort the vector into ascending order
> sort(scorev)
Angie Suzie Chris  Matt   Liz
 1.85  2.91  3.04  8.48  9.51
> # Calculate index to sort into ascending order
> order(scorev)
[1] 1 3 2 4 5
> # Sort the vector into ascending order
> scorev[order(scorev)]
Angie Suzie Chris  Matt   Liz
 1.85  2.91  3.04  8.48  9.51
> # Calculate the sorted (ordered) vector
> sortv <- scorev[order(scorev)]
> # Calculate index to sort into unsorted (original) order
> order(order(scorev))
[1] 1 3 2 4 5
> sortv[order(order(scorev))]
Angie Chris Suzie  Matt   Liz
 1.85  3.04  2.91  8.48  9.51
> scorev
Angie Chris Suzie  Matt   Liz
 1.85  3.04  2.91  8.48  9.51
> # Examples for sort() with ties
> order(c(2, 1:4))  # There's a tie
[1] 2 1 3 4 5
> order(c(2, 1:4), 1:5)  # There's a tie
[1] 2 1 3 4 5
```

Sorting Data Frames

Data frames can be sorted on any one of its columns.

```
> # Create a vector of student ranks
> rankv <- c("fifth", "fourth", "third", "second", "first")
> # Reverse sort the student ranks according to students
> rankv[order(order(scorev))]
[1] "fifth"  "third"  "fourth" "second" "first"
> # Create a data frame of students and their ranks
> rosterdf <- data.frame(score=scorev,
+   rank=rankv[order(order(scorev))])
> rosterdf
      score   rank
Angie  1.85  fifth
Chris  3.04  third
Suzie  2.91 fourth
Matt   8.48 second
Liz    9.51  first
> # Permutation index on price column
> order(dframe$price)
[1] 2 3 1
> # Sort dframe on price column
> dframe[order(dframe$price), ]
         type  color price
flower2 daisy  white   0.5
flower3 tulip yellow   1.0
flower1  rose    red   1.5
> # Sort dframe on color column
> dframe[order(dframe$color), ]
         type  color price
flower1  rose    red   1.5
flower2 daisy  white   0.5
flower3 tulip yellow   1.0
```

Coercing Data Frames Into Matrices Using as.matrix()

The function as.matrix() coerces vectors and data frames into matrices.

Coercing a data frame into a matrix causes coercion of numeric values into character.

as.matrix() coerces vectors into single column matrices, as opposed to matrix(), which produces a matrix.

```
> as.matrix(dframe)
        type    color    price
flower1 "rose"  "red"    "1.5"
flower2 "daisy" "white"  "0.5"
flower3 "tulip" "yellow" "1.0"
> vectorv <- sample(9)
> matrix(vectorv, ncol=3)
     [,1] [,2] [,3]
[1,]    2    8    4
[2,]    5    9    6
[3,]    7    1    3
> as.matrix(vectorv, ncol=3)
      [,1]
 [1,]    2
 [2,]    5
 [3,]    7
 [4,]    8
 [5,]    9
 [6,]    1
 [7,]    4
 [8,]    6
 [9,]    3
```

Coercing Matrices Into Data Frames

The generic function `as.data.frame()` coerces matrices and other objects into data frames.

The method `as.data.frame.matrix()` coerces only matrices into data frames.

`as.data.frame.matrix()` is about 50% faster than `as.data.frame()`, because it skips extra `R` code in `as.data.frame()` needed for argument validation, error checking, and method dispatch.

As a general rule, calling generic functions is slower than directly calling individual methods, because generic functions must execute extra `R` code for method dispatch.

The function `data.frame()` can also be used to coerce matrices into data frames, but is much slower than even `as.data.frame()`.

`as.data.frame()` is about three times faster than `data.frame()`, because it doesn't require extra `R` code in `data.frame()` needed for handling different types of vectors, and for method dispatch.

```
> library(microbenchmark)
> # Call method instead of generic function
> as.data.frame.matrix(matrixv)
> # A few methods for generic function as.data.frame()
> sample(methods(as.data.frame), size=4)
> # Function method is faster than generic function
> summary(microbenchmark(
+   as_dframe_matrix=
+     as.data.frame.matrix(matrixv),
+   as_dframe=as.data.frame(matrixv),
+   dframe=data.frame(matrixv),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing Matrices Into Lists

Matrices can be coerced into lists in at least two different ways.

Matrices can be first coerced into a data frame, and then into a list using function `as.list()`.

Matrices can be directly coerced into a list using function `lapply()`.

Using `lapply()` is the faster of the two methods, because `lapply()` is a *compiled* function.

```
> # lapply is faster than coercion function
> summary(microbenchmark(
+   aslist=as.list(as.data.frame.matrix(matrixv)),
+   lapply=lapply(seq_along(matrixv[1, ]),
+     function(indeks) matrixv[, indeks]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
    expr  mean median
1 aslist  9.41   7.97
2 lapply 59.45   3.24
```

The iris Data Frame

The iris data frame is included in the datasets base package.

iris contains sepal and petal dimensions of 50 flowers from 3 species of iris.

The function unique() extracts unique elements of an object.

sapply() applies a function to a list or a vector of objects and returns a vector.

sapply() performs a loop over the list of objects, and can replace "for" loops in R.

```
> # ?iris  # Get information on iris
> dim(iris)
[1] 150   5
> head(iris, 2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Spe
> unique(iris$Species)  # List of unique elements of iris
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
> class(unique(iris$Species))
[1] "factor"
> # Find which columns of iris are numeric
> sapply(iris, is.numeric)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
        TRUE         TRUE         TRUE         TRUE        FALSE
> # Calculate means of iris columns
> sapply(iris, mean)  # Returns NA for Species
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
        5.84         3.06         3.76         1.20           NA
```

The mtcars Data Frame

The mtcars data frame is included in the datasets base package, and contains design and performance data for 32 automobiles.

```
> # ?mtcars  # mtcars data from 1974 Motor Trend magazine
> # mpg   Miles/(US) gallon
> # qsec   1/4 mile time
> # hp  Gross horsepower
> # wt  Weight (lb/1000)
> # cyl   Number of cylinders
> dim(mtcars)
[1] 32 11
> head(mtcars, 2)
              mpg cyl disp  hp drat   wt qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.62 16.5  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.88 17.0  0  1    4    4
> colnames(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"
[11] "carb"
> head(rownames(mtcars), 3)
[1] "Mazda RX4"     "Mazda RX4 Wag" "Datsun 710"
> unique(mtcars$cyl)  # Extract list of car cylinders
[1] 6 4 8
> sapply(mtcars, mean)  # Calculate means of mtcars columns
    mpg     cyl    disp      hp    drat      wt    qsec      vs
 20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438   0.
   carb
  2.812
```

The Cars93 Data Frame

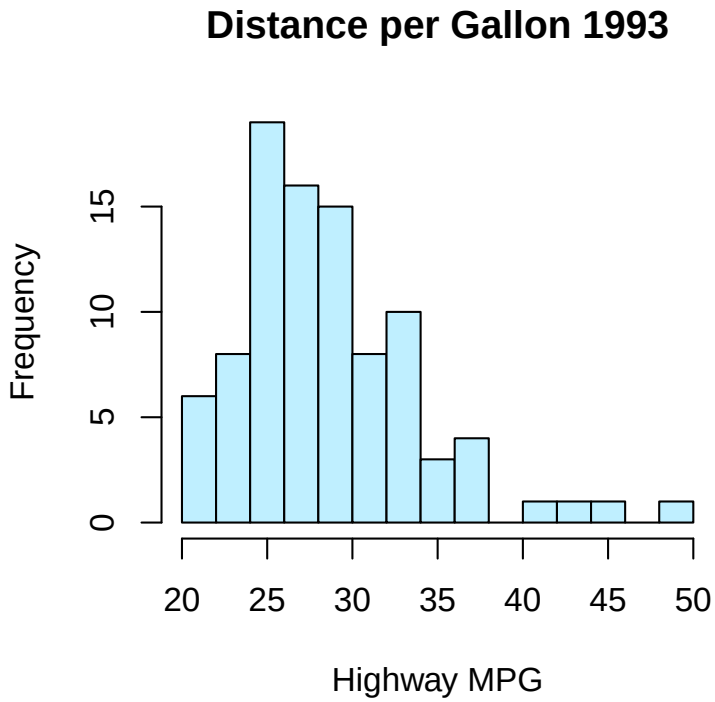
The Cars93 data frame is included in the MASS package, and contains design and performance data for 93 automobiles.

The function hist() calculates and plots a histogram, and returns its data *invisibly*.

The parameter breaks is the number of cells of the histogram.

"FD" stands for the Freedman-Diaconis rule for calculating histogram breaks,

```
> library(MASS)
> # ?Cars93  # Get information on Cars93
> dim(Cars93)
> head(colnames(Cars93))
> # head(Cars93, 2)
> unique(Cars93$Type)  # Extract list of car types
> # sapply(Cars93, mean)  # Calculate means of Cars93 columns
> # Plot histogram of Highway MPG using the Freedman-Diaconis rule
> hist(Cars93$MPG.highway, col="lightblue1",
+      main="Distance per Gallon 1993", xlab="Highway MPG", breaks="FD")
```



Types of Bad Data

Possible sources of bad data are: imported data, class coercion, numeric overflow.

Types of bad data:

- `NA` (not available) is a logical constant indicating missing data,
- `NaN` means Not a Number data,
- `Inf` means numeric overflow - divide by zero,

When a function produces `NA` or `NaN` values, then it also produces a *warning* condition, but not an *error*.

`NA` or `NaN` values are not *errors*.

The functions `is.na()` and `is.nan()` test for `NA` and `NaN` values.

Many functions have a `na.rm` parameter to remove `NA`s from input data.

```
> as.numeric(c(1:3, "a"))  # NA from coercion
[1]  1  2  3 NA
> 0/0  # NaN from ambiguous math
[1] NaN
> 1/0  # Inf from divide by zero
[1] Inf
> is.na(c(NA, NaN, 0/0, 1/0))  # Test for NA
[1]  TRUE  TRUE  TRUE FALSE
> is.nan(c(NA, NaN, 0/0, 1/0))  # Test for NaN
[1] FALSE  TRUE  TRUE FALSE
> NA*1:4  # Create vector of Nas
[1] NA NA NA NA
> # Create vector with some NA values
> datav <- c(1, 2, NA, 4, NA, 5)
> datav
[1]  1  2 NA  4 NA  5
> mean(datav)  # Returns NA, when NAs are input
[1] NA
> mean(datav, na.rm=TRUE)  # remove NAs from input data
[1] 3
> datav[!is.na(datav)]  # Delete the NA values
[1] 1 2 4 5
> sum(!is.na(datav))  # Count non-NA values
[1] 4
```

Scrubbing Bad Data

The function `complete.cases()` returns `TRUE` if a row has no `NA` values.

```
> # airquality data has some NAs
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> dim(airquality)
[1] 153   6
> # Number of NA elements
> sum(is.na(airquality))
[1] 44
> # Number of rows with NA elements
> sum(!complete.cases(airquality))
[1] 42
> # Display rows containing NAs
> head(airquality[!complete.cases(airquality), ])
   Ozone Solar.R Wind Temp Month Day
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
10    NA     194  8.6   69     5  10
11     7      NA  6.9   74     5  11
25    NA      66 16.6   57     5  25
26    NA     266 14.9   58     5  26
```

Scrubbing Data Using Carry Forward

Rows containing bad data may be either removed or replaced with an estimated value.

The function `stats::na.omit()` removes individual `NA` values from vectors, and it also removes whole rows of data containing `NA` values from matrices and data frames.

Bad data can also be replaced with the most recent prior values (carry forward good data).

The function `zoo::na.locf()` replaces `NA` values with the most recent non-`NA` values prior to it (*locf* stands for *last observation carry forward*).

Copying the last non-`NA` values forward causes less data loss than removing whole rows of data.

The function `na.locf()` with argument `fromLast=TRUE` replaces `NA` values with non-`NA` values in reverse order, starting from the end.

```
> # Create vector containing NA values
> vectorv <- sample(22)
> vectorv[sample(NROW(vectorv), 4)] <- NA
> # Replace NA values with the most recent non-NA values
> zoo::na.locf(vectorv)
 [1] 13 18 17  9 15 15 22 19 19  6 14 11  8  3 21 12  2  2  5  7 16
> # Remove rows containing NAs
> good_air <- airquality[complete.cases(airquality), ]
> dim(good_air)
[1] 111   6
> # NAs removed
> head(good_air)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
> # Another way of removing NAs
> fresh_air <- na.omit(airquality)
> all.equal(fresh_air, good_air, check.attributes=FALSE)
[1] TRUE
> # Replace NAs
> good_air <- zoo::na.locf(airquality)
> dim(good_air)
[1] 153   6
> # NAs replaced
> head(good_air)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    18     313 14.3   56     5   5
6    28     313 14.9   66     5   6
```

Scrubbing Time Series Data

Missing asset prices and returns can be replaced with the most recent prior values (carry forward good data).

But missing asset returns should not be replaced with values from the future. Instead, missing returns should be replaced with zero values.

The function `na.locf.xts()` from package *xts* is faster than `zoo::na.locf()`, but it only operates on time series of class `"xts"`.

```
> # Replace NAs in xts time series
> library(rutils)  # load package rutils
> pricev <- rutils::etfenv$prices[, 1]
> head(pricev, 3)
           DBC
1998-12-22  NA
1998-12-23  NA
1998-12-24  NA
> sum(is.na(pricev))
[1] 1790
> pricezoo <- zoo::na.locf(pricev, fromLast=TRUE)
> pricexts <- xts:::na.locf.xts(pricev, fromLast=TRUE)
> all.equal(pricezoo, pricexts, check.attributes=FALSE)
[1] TRUE
> head(pricexts, 3)
           DBC
1998-12-22  22
1998-12-23  22
1998-12-24  22
> library(microbenchmark)
> summary(microbenchmark(
+   zoo=zoo::na.locf(pricev, fromLast=TRUE),
+   xts=xts:::na.locf.xts(pricev, fromLast=TRUE),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
  expr mean median
1  zoo 26.3   25.6
2  xts 29.0   24.5
```

NULL Values

NULL represents a null object, and is a legitimate value, not bad data.

NULL is often returned by functions whose value is undefined.

NULL can also be used to initialize vectors.

NULL is not the same as NA values or zero-length (empty) vectors.

The functions numeric() and character() return empty (zero-length) vectors of the specified *type*.

The function is.null() tests for NULL values.

Very often variables are initialized to NULL before the start of iteration.

A more efficient way to perform iteration is by pre-allocating the vector.

```
> # NULL values have no mode or type
> c(mode(NULL), mode(NA))
[1] "NULL"    "logical"
> c(typeof(NULL), typeof(NA))
[1] "NULL"    "logical"
> c(length(NULL), length(NA))
[1] 0 1
> # Check for NULL values
> is.null(NULL)
[1] TRUE
> # NULL values are ignored when combined into a vector
> c(1, 2, NULL, 4, 5)
[1] 1 2 4 5
> # But NA value isn't ignored
> c(1, 2, NA, 4, 5)
[1]  1  2 NA  4  5
> # Vectors can be initialized to NULL
> vectorv <- NULL
> is.null(vectorv)
[1] TRUE
> # Grow the vector in a loop - very bad code!!!
> for (indeks in 1:5)
+   vectorv <- c(vectorv, indeks)
> # Initialize empty vector
> vectorv <- numeric()
> # Grow the vector in a loop - very bad code!!!
> for (indeks in 1:5)
+   vectorv <- c(vectorv, indeks)
> # Allocate vector
> vectorv <- numeric(5)
> # Assign to vector in a loop - good code
> for (indeks in 1:5)
+   vectorv[indeks] <- runif(1)
```