Control Statements, Operators and Debugging

FRE6871 & FRE7241, Spring 2023

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

May 12, 2023

Logical Operators

R has the following logical operators:

- "<" less than,
- "<=" less than or equal to,
- ">" greater than,
- ">=" greater than or equal to,
- "==" exactly equal to,
- "!=" not equal to,
- "!x" Not x,
- "x & y" x AND y,
- "x | y" x OR y,

These operators are applied to vectors element-wise.

```
> TRUE | FALSE
> TRUE | NA
> vector1 <- c(2, 4, 6)
> vector1 < 5  # Element-wise comparison
> (vector1 < 5) & (vector1 > 3)
> vector1[(vector1 < 5) & (vector1 > 3)]
> vector2 <- c(-10, 0, 10)
> vector1 < vector2
> c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)
> c(FALSE, TRUE, FALSE) | c(TRUE, TRUE, FALSE)
```

Long Form Logical Operators

R also has two long form logical operators:

- "x && y" x AND y,
- "x || y" x OR y,

These operators differ from the short form operators in two ways:

- They only evaluate the first elements of their vector arguments,
- They short-circuit (stop evaluation as soon as the expression is determined),

Rule of Thumb

- Use "&&" and "||" in if-clauses,

```
> c(FALSE, TRUE, FALSE) && c(TRUE, TRUE, FALSE)
> c(FALSE, TRUE, FALSE) || c(TRUE, TRUE, FALSE)
> echo_true <- function() {cat("echo_true\t"); TRUE}
> echo_false <- function() {cat("echo_false\t"); FALSE}
> echo_true() | echo_false()
> echo_true() || echo_false()  # echo_false() isn't evaluated at all
> vectorv <- c(2, 4, 6)
> # Works (does nothing) using '&&'
> if (is.matrix(vectorv) && (vectorv[2, 3] > 0)) {
+   vectorv[2, 3] <- 1
+ }
> # No short-circuit so fails (produces an error)
> if (is.matrix(vectorv) & (vectorv[2, 3] > 0)) {
+   vectorv[2, 3] <- 1
+ }
```

Arithmetic Operators

Arithmetic *operators* perform arithmetic operations on `numeric` or `complex` vectors,

- "`+`" performs addition,
- "`-`" performs subtraction,
- "`*`" performs multiplication,
- "`/`" performs division,
- "`^`" and "`**`" perform exponentiation,

```
> ?Arithmetic
> 4.7 * 0.5  # Multiplication
> 4.7 / 0.5  # Division
> # Exponentiation
> 2**3
> 2^3
```

Comparing Objects With identical() and all.equal()

The function identical() tests if two objects are exactly the same, and always returns a single logical TRUE or FALSE (never NA or logical vectors).

For atomic arguments identical() often gives the same result as the "==" operator, but it's not synonymous with it in general.

The "==" operator applies the *recycling rule* to vector arguments and returns logical vectors, but identical() doesn't and returns a single logical value.

The function all.equal() tests the equality of two objects to within the square root of the *machine precision*.

The variable .Machine contains information about the numerical characteristics of the computer R is running on, such as the largest double and integer numbers, and the *machine precision*.

```
> numv <- 2
> numv==2
> identical(numv, 2)
>
> identical(numv, NULL)
> # This doesn't work:
> # numv==NULL
> is.null(numv)
>
> vectorv <- c(2, 4, 6)
> vectorv==2
> identical(vectorv, 2)
>
> # numv is equal to "1.0" within machine precision
> numv <- 1.0 + 2*sqrt(.Machine$double.eps)
> all.equal(numv, 1.0)
>
> # Info machine precision of computer R is running on
> # ?.Machine
> # Machine precision
> .Machine$double.eps
```

Lookup and Matching Using which() and match()

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

If the argument is an array and arr.ind=TRUE, then which() returns a matrix with rows containing the indices of the TRUE elements.

The functions which.max() and which.min() return the index of the minimum or maximum of a numeric or Boolean vector.

match() returns the index of the vector element that *exactly* matches its first argument.

If it doesn't find an exact match then it returns NA.

The expressions match(x, vectorv) and min(which(vectorv == x)) produce the same result, but match() can be faster for large vectors.

```
> vectorv <- sample(1e3, 1e3)
> matrixv <- matrix(vectorv, ncol=4)
> which(vectorv == 5)
> match(5, vectorv)
> # Equivalent but slower than above
> (1:NROW(vectorv))[vectorv == 5]
> which(vectorv < 5)
> # Find indices of TRUE elements of Boolean matrix
> which((matrixv == 5)|(matrixv == 6), arr.ind=TRUE)
> # Equivalent but slower than above
> arrayInd(which((matrixv == 5)|(matrixv == 6)),
+    dim(matrixv), dimnames(matrixv))
> # Find index of largest element
> which.max(vectorv)
> which(vectorv == max(vectorv))
> # Find index of smallest element
> which.min(vectorv)
> # Benchmark match() versus which()
> all.equal(match(5, vectorv), min(which(vectorv == 5)))
> library(microbenchmark)
> summary(microbenchmark(
+   match=match(5, vectorv),
+   which=min(which(vectorv == 5)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Lookup and Matching Using %in% and any()

The binary operator %in% returns a Boolean vector with TRUE values corresponding to elements that have matches.

%in% is a wrapper for match() defined as follows: "%in%" <- function(x, table) match(x, table, nomatch=0) > 0.

%in% never returns NA, so it's preferred in if() statements.

any() returns TRUE if at least one element of a Boolean vector is TRUE, and FALSE otherwise.

The function pmatch() performs partial matching of strings.

```
> # Does 5 belong in vectorv?
> 5 %in% vectorv
> match(5, vectorv, nomatch=0) > 0
> # Does (-5) belong in vectorv?
> (-5) %in% vectorv
> c(5, -5) %in% vectorv
> match(-5, vectorv)
> # Equivalent to "5 %in% vectorv"
> any(vectorv == 5)
> # Equivalent to "(-5) %in% vectorv"
> any(vectorv == (-5))
> # Any negative values in vectorv?
> any(vectorv < 0)
> # Example of use in if() statement
> if (any(vectorv < 2))
+   cat("vector contains small values\n")
> # Partial matching of strings
> pmatch("med", c("mean", "median", "mode"))
```

Finding Closest Match Using findInterval()

The function match() returns the index of the vector element that *exactly* matches its first argument.

If match() doesn't find an exact match then it returns NA.

The function 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".

```
> # Display the formal arguments of findInterval
> args(findInterval)
> # Get index of the element of "vec" that matches 5
> findInterval(x=5, vec=c(3, 5, 7))
> match(5, c(3, 5, 7))
> # No exact match
> findInterval(x=6, vec=c(3, 5, 7))
> match(6, c(3, 5, 7))
> # Indices of "vec" that match elements of "x"
> findInterval(x=1:8, vec=c(3, 5, 7))
> # Return only indices of inside intervals
> findInterval(x=1:8, vec=c(3, 5, 7), all.inside=TRUE)
> # Make rightmost interval inclusive
> findInterval(x=1:8, vec=c(3, 5, 7), rightmost.closed=TRUE)
```

Assignment Operators

The standard assignment operator in R is "<-".

Both "<-" and "=" are valid assignment operators in R.

The "<-" operator may cause an error if R confuses it with the "<" logical operator.

But they differ in *scope* and *precedence* ("<-" has higher precedence than "=").

The "=" operator is used for named arguments in function calls.

When variables are assigned within an argument list using the "=" operator, their *scope* is limited to the function.

Rule of Thumb:
Use "<-" in R scripts and inside functions,
Use "=" only in function calls.

```
> numv1 <- 3  # "<-" and "=" are valid assignment operators
> numv1
> numv1 = 3
> numv1
> 2<-3  # "<" operator confused with "<-"
> 2 < -3  # Add space or brackets to avoid confusion
> # "=" assignment within argument list
> median(x=1:10)
> x  # x doesn't exist outside the function
> # "<-" assignment within argument list
> median(x <- 1:10)
> x  # x exists outside the function
```

The assign() Function

The assign() function assigns a value to an object in a specified *environment*, by referencing it using a character string (name).

assign() can be used to either assign values to existing variables, or to create new variables.

assign() looks for the object name in the specified *environment*, and assigns a value to it.

If assign() can't find the object name, then it creates it.

assign() expects a character string as its argument.

If a object name is passsed to assign(), then it evaluates that object to get the string it contains.

If the object doesn't contain a string, then assign() produces an error.

```
> myvar <- 1  # Create new object
> assign(x="myvar", value=2)  # Assign value to existing object
> myvar
> rm(myvar)  # Remove myvar
> assign(x="myvar", value=3)  # Create new object from name
> myvar
> # Create new object in new environment
> new_env <- new.env()  # Create new environment
> assign("myvar", 3, envir=new_env)  # Assign value to name
> ls(new_env)  # List objects in "new_env"
> new_env$myvar
> rm(list=ls())  # Delete all objects
> symbol <- "myvar"  # Define symbol containing string "myvar"
> assign(symbol, 1)  # Assign value to "myvar"
> ls()
> myvar
> assign("symbol", "new_var")
> assign(symbol, 1)  # Assign value to "new_var"
> ls()
> symbol <- 10
> assign(symbol, 1)  # Can't assign to non-string
```

Applying assign() to Lists of Names

assign() allows creating new objects from listv or vectors of names (character strings), such as column names.

```
> rm(list=ls())  # Delete all objects
> # Create individual vectors from column names of EuStockMarkets
> for (colname in colnames(EuStockMarkets)) {
+ # Assign column values to column names
+   assign(colname, EuStockMarkets[, colname])
+ }  # end for
> ls()
> head(DAX)
> head(EuStockMarkets[, "DAX"])
> identical(DAX, EuStockMarkets[, "DAX"])
```

Retrieving Objects Using get()

The function `get()` accepts a character string and returns the value of the corresponding object in a specified *environment*.

`get()` retrieves objects that are referenced using character strings, instead of their names.

The functions `get()` and `assign()` allow retrieving and assigning values to objects that are referenced using character strings.

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects.

```
> # Create new environment
> test_env <- new.env()
> # Pass string as name to create new object
> assign("myvar1", 2, envir=test_env)
> # Create new object using $ string referencing
> test_env$myvar2 <- 1
> # List objects in new environment
> ls(test_env)
> # Reference an object by name
> test_env$myvar1
> # Reference an object by string name using get
> get("myvar1", envir=test_env)
> # Retrieve and assign value to object
> assign("myvar1",
+        2*get("myvar1", envir=test_env),
+        envir=test_env)
> get("myvar1", envir=test_env)
> # Return all objects in an environment
> mget(ls(test_env), envir=test_env)
> # Delete environment
> rm(test_env)
```

Metaprogramming in R

A powerful feature of R is *non-standard evaluation* (aka *metaprogramming* or *programming on the language*).

Unevaluated *expressions* are objects that represent R formulas and commands.

Metaprogramming allows creating and manipulating unevaluated R *expressions* and then executing them as needed.

The book *Advanced R* by Hadley Wickham provides a good explanation of *metaprogramming* in R and R *expressions*.

R interretsp character strings that are not in quotes "" as *symbols* or *expressions*.

The function `as.symbol()` converts a character string into a *symbol* object.

The function `parse()` converts a character string into an unevaluated *expression* object.

The function `eval()` evaluates a *symbol* or *expression* in a specified *environment*.

```
> rm(list=ls())  # Delete all objects
> # Convert string to symbol
> as.symbol("some_string")
> # The "name" class is synonymous with a symbol
> class(as.symbol("some_string"))
> # Symbols are created during assignments
> symbol <- 2
> # Evaluate symbol (same as typing it)
> eval(symbol)
> # Convert string into a symbol and evaluate it
> eval(as.symbol("symbol"))
> # Convert string into unevaluated expression
> expv <- parse(text="newv <- symbol")
> expv
> class(expv)
> ls()
> eval(expv)  # Evaluate expression
> ls()  # Expression evaluation created new object
> newv
```

Manipulating Symbols and Expressions

The function `quote()` accepts *symbols* and *expressions*, and returns an *expression* object without evaluating it.

The function `quote()` creates unevaluated *expression* objects which later can be evaluated by functions.

The function `substitute()` replaces objects in unevaluated expressions with their corresponding values, and returns an *expression*.

`substitute()` looks up the object names in either named `listv` (symbol-value pairs) or in `environments`, and evaluates the objects in them.

`substitute()` is often used inside functions to substitute formal arguments with the names of the actual arguments they are bound to in a function call.

```
> # Create the expression "1+1"
> quote(1+1)
> # Evaluate the expression "1+1"
> eval(quote(1+1))
> # Create an expression containing several commands
> expv <- quote({x <- 1; y <- 2; x+y})
> expv
> # Evaluate all the commands in the expression
> eval(expv)
> ls()
> # Return an expression without evaluating it
> newv <- 2*symbol
> expv <- quote(symbol + newv)
> expv
> eval(expv)  # Evaluate expression
> # Substitute objects in an expression
> expv <- substitute(symbol + newv,
+                env=list(symbol=1, newv=2))
> expv
> eval(expv)  # Evaluate expression
> # Get_input() substitutes its formal argument with the actual argu
> get_input <- function(input) {
+   substitute(input)
+ }  # end get_input
> myvar <- 2
> get_input(myvar)
> eval(get_input(myvar))
```

Converting Symbols and Expressions Into Strings

The function `deparse()` is the opposite of `parse()`, and it converts *symbols* and *expressions* into character strings.

The combination of functions `deparse(substitute())` returns a character string representing the actual argument passed into a function.

```
> # Define symbol
> myvar <- 10
> # Convert symbol value into string
> deparse(myvar)
> # Convert symbol into string without evaluating it
> deparse(quote(myvar))
> # Substitute object with value from named list
> symbol <- 2
> deparse(substitute(symbol + myvar,
+         env=list(myvar=2)))
> # Create string with name of input argument
> get_name <- function(input) {
+   names(input) <- deparse(substitute(input))
+   input
+ }  # end get_name
> get_name(myvar)
```

The Parenthesis "()" and Curly Braces "{}" Operators

The parenthesis "()" and curly braces "{}" operators are used to enclose and to group (combine) expressions.

The parenthesis "()" and curly braces "{}" operators are functions, and they return values.

An expression enclosed by the parenthesis "()" operator is evaluated separatately from other expressions, and its result is returned.

Enclosing expressions in parenthesis makes them less ambiguous.

The curly braces "{}" operator can group several expressions, that can be written either on separate lines, or be separated by the semicolon ";" operator.

The curly braces "{}" operator returns the last expression it encloses.

Both the parenthesis "()" and curly braces "{}" operators are functions, and executing them requires a little additional processing time.

The square braces (brackets) "[]" operator subsets (references) the elements of vectors, matrices, and listv.

```
> # Expressions enclosed in parenthesis are less ambiguous
> -2:5
> (-2):5
> -(2:5)
> # Expressions enclosed in parenthesis are less ambiguous
> -2*3+5
> -2*(3+5)
>
> # Expressions can be separated by semicolons or by lines
> {1+2; 2*3; 1:5}
> # or
> {1+2
+ 2*3
+ 1:5}
>
> matrixv <- matrix(nr=3, nc=4)
> matrixv <- 0
> # Subset whole matrix
> matrixv[] <- 0
>
> # Parenthesis and braces require a little additional processing ti
> library(microbenchmark)
> summary(microbenchmark(
+   basep=sqrt(rnorm(10000)^2),
+   parven=sqrt(((((rnorm(10000)^2)))))),
+   bra_ce=sqrt({{{{rnorm(10000)^2}}}}),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

The "if () else" Control Statement

R has the familiar "if () {...} else {...}" statement to control execution flow depending on logical conditions.

The logical conditions must be either a Boolean or numeric type, otherwise an error is produced.

The "else" statement can also be omitted.

"if" statements can be nested using multiple "else if" statements.

```
> numv1 <- 1
>
> if (numv1) {  # Numeric zero is FALSE, all other numbers are TRUE
+   numv2 <- 4
+ } else if (numv1 == 0) {  # 'else if' together on same line
+   numv2 <- 0
+ } else {  # 'else' together with curly braces
+   numv2 <- -4
+ }  # end if
>
> numv2
```

The switch() Control Statement

The function switch() matches its first argument "EXPR" with one of the symbols in the following arguments, evaluates the corresponding expression, and returns it.

The arguments that follow the first argument "EXPR" should be given as *symbol=value* pairs.

If "EXPR" is a character string, then the expression bound to that symbol is returned by switch().

If "EXPR" is an integer, then switch() returns the expression from that position.

If switch() can't match "EXPR" to any symbol, then it returns NULL invisibly.

Using switch() is a convenient alternative to a cascade of "if () else" statements.

The function match.arg() matches a string to one of the possible values, and returns the matched value, or produces an error if it can't match it.

```
> switch("a", a="aaahh", b="bee", c="see", d=2,
+        "else this")
> switch("c", a="aaahh", b="bee", c="see", d=2,
+        "else this")
> switch(3, a="aaahh", b="bee", c="see", d=2,
+        "else this")
> switch("cc", a="aaahh", b="bee", c="see", d=2,
+        "else this")
> # Measure of central tendency
> centra_lity <- function(input,
+     method=c("mean", "mean_narm", "median")) {
+ # validate "method" argument
+   method <- match.arg(method)
+   switch(method,
+  mean=mean(input),
+  mean_narm=mean(input, na.rm=TRUE),
+  median=median(input))
+ }  # end centra_lity
> myvar <- rnorm(100, mean=2)
> centra_lity(myvar, "mean")
> centra_lity(myvar, "mean_narm")
> centra_lity(myvar, "median")
```

Iteration Using for() and while() Loops

The for() loop statement:

```
> for (indeks in vectorv) {expvs}
```

iterates the *dummy* variable indeks over the elements of the vector or list colorl, and evaluates in a loop the ex_pressions contained in the body of the for() loop.

Upon loop exit the *dummy* variable indeks is left equal to the last element of the vector colorl.

while() loops start by testing their logical condition, and they repeat executing the loop body until that condition is FALSE.

But while() loops risk producing infinite loops if not written properly, so Use Them With Care!

```
> colorl <- list("red", "white", "blue")
> # Loop over list
> for (some_color in colorl) {
+   print(some_color)
+ }  # end for
> # Loop over vector
> for (indeks in 1:3) {
+   print(colorl[[indeks]])
+ }  # end for
>
> # While loops require initialization
> indeks <- 1
> # While loop
> while (indeks < 4) {
+   print(colorl[[indeks]])
+   indeks <- indeks + 1
+ }  # end while
```

Performing Loops Using for() and apply()

The for() loop doesn't return a value, so values calculated in the for() loop body must be assigned to variables in the parent environment, or otherwise they are lost.

The expressions in the for() loop body have access to variables in the parent environment in which the for() loop is executed, and they can modify those variables.

So even though for() loops don't return a value, they can be used to perform calculations on variables in the parent environment, but this is discouraged since it can produce errors that are hard to debug.

Rule of Thumb:

- for() loops are preferred for producing *side effects*, like plotting or reading and writing data to files,
- apply() loops are preferred for performing calculations which produce vectors or matrices of values.

```
> vectorv <- integer(7)
> # Loop over a vector and overwrite it
> for (i in seq_along(vectorv)) {
+   cat("Changing element:", i, "\n")
+   vectorv[i] <- i^2
+ }  # end for
> # Modifying vectorv inside sapply() has no effect
> vectorv <- integer(7)
> vectorv
> sapply(seq_along(vectorv),
+  function(i) {
+    vectorv[i] <- i^2
+  })  # end sapply
> vectorv
> # Super-assignment operator "<<-" allows modifying vectorv
> sapply(seq_along(vectorv),
+  function(i) {
+    vectorv[i] <<- i^2 # "<<-" !!!
+  })  # end sapply
> vectorv
> # sapply() loop returns vector of values
> vectorv <- sapply(seq_along(vectorv), function(i) (i^2))
```

Fibonacci Sequence Using for() Loop

The *Fibonacci* sequence of integers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

$$F_1 = 0, F_2 = 1,$$

$$F_n = 0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

The *Fibonacci* sequence was invented by the *Indian* mathematician Virahanka in the 8th century AD, and later described by the Italian mathematician *Fibonacci* in his famous treatise *Liber Abaci*.

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

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

The function `numeric()` returns an zero length `numeric` vector.

The function `numeric(k)` returns a `numeric` vector of zeros of length `k`.

```
> # fib_seq <- numeric()  # zero length numeric vector
> # Pre-allocate vector instead of "growing" it
> fib_seq <- numeric(10)
> fib_seq[1] <- 0  # Initialize
> fib_seq[2] <- 1  # Initialize
> for (i in 3:10) {  # Perform recurrence loop
+   fib_seq[i] <- fib_seq[i-1] + fib_seq[i-2]
+ }  # end for
> fib_seq
```

Allocating Memory to Vectors and Matrices

`R` automatically allocates memory to new objects as needed during runtime, but at the cost of slowing down calculations.

Allocating memory of the correct *mode* speeds up calculations by avoiding automatic memory allocation by `R`.

The functions `character()`, `integer()`, and `numeric()` return zero-length vectors of the specified *mode*.

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

The function `character(k)` returns a `character` vector of empty strings of length `k`.

The function `integer(k)` returns a `integer` vector of zeros of length `k`.

The function `numeric(k)` returns a `numeric` vector of zeros of length `k`.

The function `vector()` by default returns a `Boolean` vector, unless the *mode* is specified.

The function `matrix()` by default returns a `Boolean` matrix containing `NA` values, unless the *mode* is specified.

```
> # Allocate character vector
> character()
> character(5)
> is.character(character(5))
> # Allocate integer vector
> integer()
> integer(5)
> is.integer(integer(5))
> is.numeric(integer(5))
> # Allocate numeric vector
> numeric()
> numeric(5)
> is.integer(numeric(5))
> is.numeric(numeric(5))
> # Allocate Boolean vector
> vector()
> vector(length=5)
> # Allocate numeric vector
> vector(length=5, mode="numeric")
> is.null(vector())
> # Allocate Boolean matrix
> matrix()
> is.null(matrix())
> # Allocate integer matrix
> matrix(NA_integer_, nrow=3, ncol=2)
> is.integer(matrix(NA_integer_, nrow=3, ncol=2))
> # Allocate numeric matrix
> matrix(NA_real_, nrow=3, ncol=2)
> is.numeric(matrix(NA_real_, nrow=3, ncol=2))
```

Logical Operators Applied to Vectors and Matrices

When logical operators are applied to vectors and matrices, they are applied element-wise, producing `Boolean` vectors and matrices.

```
> vectorv <- sample(1:9)
> vectorv
> vectorv < 5  # Element-wise comparison
> vectorv == 5  # Element-wise comparison
> matrixv <- matrix(vectorv, ncol=3)
> matrixv
> matrixv < 5  # Element-wise comparison
> matrixv == 5  # Element-wise comparison
```

Coercing Vectors Into Matrices

Vectors can be coerced into matrices by adding a dimension attribute.

The `dimnames` attribute can be assigned a named list to convert it into a named matrix.

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

```
> matrixv <- 1:6  # Create a vector
> class(matrixv)  # Get its class
> # Is it vector or matrix?
> c(is.vector(matrixv), is.matrix(matrixv))
> structure(matrixv, dim=c(2, 3))  # Matrix object
> # Adding dimension attribute coerces into matrix
> dim(matrixv) <- c(2, 3)
> class(matrixv)  # Get its class
> # Is it vector or matrix?
> c(is.vector(matrixv), is.matrix(matrixv))
> # Assign dimnames attribute
> dimnames(matrixv) <- list(rows=c("row1", "row2"),
+             columns=c("col1", "col2", "col3"))
> matrixv
```

Coercing Matrices Into Other Types

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

But coercion functions strip the *attributes* from an object.

```
> matrixv <- matrix(1:10, 2, 5)  # Create matrix
> matrixv
> # as.numeric strips dim attribute from matrix
> as.numeric(matrixv)
> # Explicitly coerce to "character"
> matrixv <- as.character(matrixv)
> c(typeof(matrixv), mode(matrixv), class(matrixv))
> # Coercion converted matrix to vector
> c(is.matrix(matrixv), is.vector(matrixv))
```

Binding Vectors and Matrices Together

Vectors can be bound into matrices using the functions `cbind()` and `rbind()`.

The *recycling rule* allows operations on vectors of different lengths:

1. Vectors are scanned from left to right,
2. Shorter vectors are extended in length by recycling their values until they match the length of longer vectors,

```
> vector1 <- 1:3  # Define vector
> vector2 <- 6:4  # Define vector
> # Bind vectors into columns
> cbind(vector1, vector2)
> # Bind vectors into rows
> rbind(vector1, vector2)
> # Extend to four elements
> vector2 <- c(vector2, 7)
> # Recycling rule applied
> cbind(vector1, vector2)
> # Another example of recycling rule
> 1:6 + c(10, 20)
```

Replicating Objects Using rep()

The function rep() replicates vectors and listv a given number of times.

rep() accepts a vector or list "x", and an integer specifying the type and number of replications.

Argument "times" replicates the whole vector a given number of times.

Argument "each" replicates each vector element a given number of times.

Argument "length.out" replicates the whole vector a certain number of times, so that the output vector length is equal to "length.out".

```
> # Replicate a single element
> rep("a", 5)
> # Replicate the whole vector several times
> rep(c("a", "b"), 5)
> rep(c("a", "b"), times=5)
> # Replicate the first element, then the second, etc.
> rep(c("a", "b"), each=5)
> # Replicate to specified length
> rep(c("a", "b"), length.out=5)
```

Multiplying Vectors and Matrices

The multiplication "*" *operator* performs *element-wise* (*element-by-element*) multiplication of vectors and matrices.

By default the matrix elements are multiplied column-wise by the vector elements: the first matrix element in the first column is multiplied by the first vector element, then the second matrix column is multiplied by the remaining vector elements, etc.

The *recycling rule* is applied to the vector elements as needed.

The transpose function `t()` can be applied if we want to perform row-wise multiplication.

But the transpose function `t()` is very slow for large matrices.

A better choice is to use functions `lapply()` and `do.call()`.

```
> # Define vector and matrix
> vector1 <- c(2, 4, 3)
> matrixv <- matrix(sample(1:12), ncol=3)
> # Multiply columns of matrix by vector
> vector1*matrixv
> # Or
> matrixv*vector1
> # Multiply rows of matrix by vector
> t(vector1*t(matrixv))
> # Multiply rows of matrix by vector - transpose is very slow
> matrixp <- lapply(1:NCOL(matrixv),
+   function(x) vector1[x]*matrixv[, x])
> do.call(cbind, matrixp)
> library(microbenchmark)
> summary(microbenchmark(
+   trans=t(vector1*t(matrixv)),
+   lapp={
+     matrixp <- lapply(1:NCOL(matrixv), function(x) vector1[x]*matr
+     do.call(cbind, matrixp)
+   },
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Matrix Inner Multiplication

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices.

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so that each pair produces a single number:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Inner multiplication produces a vector or matrix with a reduced dimension.

Inner multiplication requires the dimensions of the matrices to be *conformable* (number of columns in the first matrix must be equal to the number of rows in the second).

The function `drop()` removes any extra dimensions of length *one*.

The functions `rowSums()` and `colSums()` calculate the sums of rows and columns, and they're very fast because they pass their data to compiled `C++` code.

```
> vector2 <- 6:4  # Define vector
> # Multiply two vectors element-by-element
> vector1 * vector2
> # Calculate inner product
> vector1 %*% vector2
> # Calculate inner product and drop dimensions
> drop(vector1 %*% vector2)
> # Multiply columns of matrix by vector
> matrixv %*% vector1  # Single column matrix
> drop(matrixv %*% vector1)  # vector
> rowSums(t(vector1 * t(matrixv)))
> # using rowSums() and t() is 10 times slower than %*%
> library(microbenchmark)
> summary(microbenchmark(
+   inner=drop(matrixv %*% vector1),
+   transp=rowSums(t(vector1 * t(matrixv))),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Matrix Transpose

The function `t()` returns the transpose of a matrix.

The function `crossprod()` also performs *inner* (*scalar*) multiplication, exactly the same as the %*% operator, but is slightly faster.

```
> # Multiply matrix by vector fails because dimensions aren't confor
> vector1 %*% matrixv
> # Works after transpose
> drop(vector1 %*% t(matrixv))
> # Calculate inner product
> crossprod(vector1, vector2)
> # Create matrix and vector
> matrixv <- matrix(1:3000, ncol=3)
> tmatrixv <- t(matrixv)
> vectorv <- 1:3
> # crossprod() is slightly faster than "%*%" operator
> summary(microbenchmark(
+   cross_prod=crossprod(tmatrixv, vectorv),
+   inner_prod=matrixv %*% vectorv,
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Matrix Outer Multiplication

An *outer* product consists of all possible products of pairs of elements of two objects:

$$C_{i,j} = A_i \cdot B_j$$

An *outer* product of a function consists of applying it to all possible pairs of elements of two objects:

$$C_{i,j} = f(A_i, B_j)$$

Outer multiplication produces an object with dimension equal to the sum of the factors' dimensions, and with the number of elements equal to the product of the factors' elements.

The function `outer()` calculates the *outer* product of two matrices, and by default multiplies the elements of its arguments.

`outer()` can also calculate the values of a vectorized function of two variables passed to the `"FUN"` argument.

```
> # Define named vectors
> vector1 <- sample(1:4)
> names(vector1) <- paste0("row", 1:4, "=", vector1)
> vector1
> vector2 <- sample(1:3)
> names(vector2) <- paste0("col", 1:3, "=", vector2)
> vector2
> # Calculate outer product of two vectors
> matrixv <- outer(vector1, vector2)
> matrixv
> # Calculate vectorized function spanned over two vectors
> matrixv <- outer(vector1, vector2,
+            FUN=function(x1, x2) x2*sin(x1))
> matrixv
```

Flattening a List of Vectors to a Matrix Using do.call()

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

If the list contains vectors of different lengths, then `R` applies the recycling rule.

If the list contains a `NULL` element, that element is skipped.

```
> # Create list of vectors
> listv <- lapply(1:3, function(x) sample(6))
> # Bind list elements into matrix - doesn't work
> rbind(listv)
> # Bind list elements into matrix - tedious
> rbind(listv[[1]], listv[[2]], listv[[3]])
> # Bind list elements into matrix - works!
> do.call(rbind, listv)
> # Create numeric list
> listv <- list(1, 2, 3, 4)
> do.call(rbind, listv)  # Returns single column matrix
> do.call(cbind, listv)  # Returns single row matrix
> # Recycling rule applied
> do.call(cbind, list(1:2, 3:5))
> # NULL element is skipped
> do.call(cbind, list(1, NULL, 3, 4))
> # NA element isn't skipped
> do.call(cbind, list(1, NA, 3, 4))
```

Efficient Binding of Lists Into Matrices

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

But for large vectors this procedure can be very slow, and often causes an out of memory error.

The function `do_call_rbind()` efficiently combines a list of vectors into a matrix.

`do_call_rbind()` produces the same result as `do.call(rbind, list_var)`, but using recursion.

`do_call_rbind()` calls lapply in a loop, each time binding neighboring list elements and dividing the length of the list by half.

`do_call_rbind()` is the same function as `do.call.rbind()` from package *qmao*:
https://r-forge.r-project.org/R/?group_id=1113

```
> list_vectors <- lapply(1:5, rnorm, n=10)
> matrixv <- do.call(rbind, list_vectors)
> dim(matrixv)
> do_call_rbind <- function(listv) {
+   while (NROW(listv) > 1) {
+ # Index of odd list elements
+     odd_index <- seq(from=1, to=NROW(listv), by=2)
+ # Bind odd and even elements, and divide listv by half
+     listv <- lapply(odd_index, function(indeks) {
+ if (indeks==NROW(listv)) return(listv[[indeks]])
+ rbind(listv[[indeks]], listv[[indeks+1]])
+     })  # end lapply
+   }  # end while
+ # listv has only one element - return it
+   listv[[1]]
+ }  # end do_call_rbind
> identical(matrixv, do_call_rbind(list_vectors))
```

Filtering Data Frames Using subset()

Filtering means extracting rows from a *data frame* that satisfy a logical condition.

Data frames can be filtered using Boolean vectors and brackets "[]" operators.

The function subset() filters *data frames*, by applying logical conditions to its columns, using the column names.

subset() provides a succinct notation and discards NA values, but it's slightly slower than using Boolean vectors and brackets "[]" operators.

```
> airquality[(airquality$Solar.R > 320 &
+         !is.na(airquality$Solar.R)), ]
> subset(x=airquality, subset=(Solar.R > 320))
> summary(microbenchmark(
+     subset=subset(x=airquality, subset=(Solar.R > 320)),
+     brackets=airquality[(airquality$Solar.R > 320 &
+             !is.na(airquality$Solar.R)), ],
+ times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Splitting Data Frames Using factor Categorical Variables

The function split() divides an object into a list of objects, according to a factor (categorical variable).

The list's namesv attribute is equal to the factor levels.

```
> unique(iris$Species)  # Species has three distinct values
> # Split into separate data frames by hand
> setosa <- iris[iris$Species=="setosa", ]
> versi <- iris[iris$Species=="versicolor", ]
> virgin <- iris[iris$Species=="virginica", ]
> dim(setosa)
> head(setosa, 2)
> # Split iris into list based on Species
> splitiris <- split(iris, iris$Species)
> str(splitiris, max.confl=1)
> names(splitiris)
> dim(splitiris$setosa)
> head(splitiris$setosa, 2)
> all.equal(setosa, splitiris$setosa)
```

The *split-apply-combine* Procedure

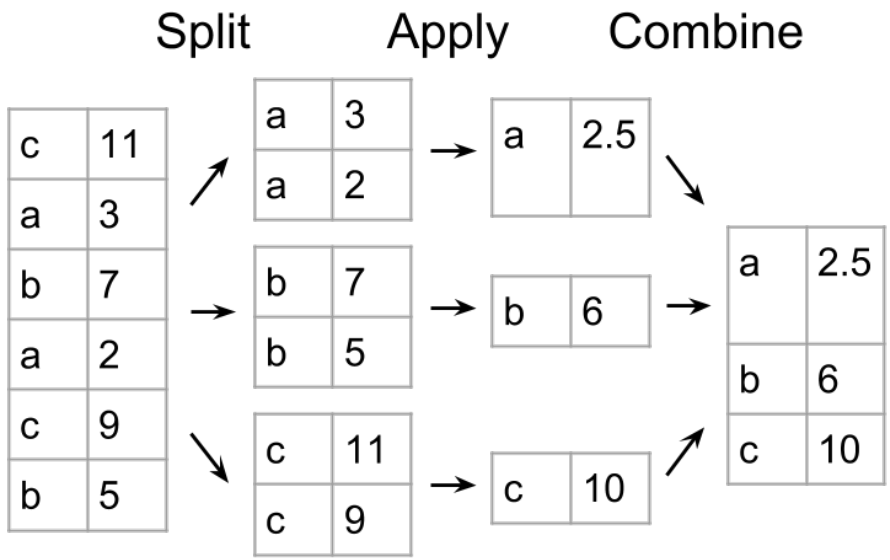
The *split-apply-combine* procedure consists of:

- dividing an object into a list, according to a factor (attribute).
- applying a function to each list element.
- combining the results.

The *split-apply-combine* procedure is also called the *map-reduce* procedure, or simply *data pivoting*, and it's similar to *pivot tables* in *Excel*.

Data pivoting can be performed *data frames*, by aggregating its columns based on categorical data stored in one of its columns.

You can read more about the *split-apply-combine* procedure in Hadley Wickham's paper:
http://www.jstatsoft.org/v40/i01/paper



Data Pivoting Example

Data pivoting can be performed through successive applications of functions `split()`, `apply()`, and `unlist()`.

A *data frame* can be *pivoted* either by first splitting it into a list of *data frames* and then aggregating, or by splitting just a single column and aggregating it.

The function `split()` divides an object into a list of objects, according to a `factor` (categorical variable).

The list's `namesv` attribute is equal to the `factor` levels.

The functional `aggregate()` *pivots* the columns of a *data frame*.

`aggregate()` can accept a `"formula"` argument with the column names, or it can accept `"x"` and `"by"` arguments with the columns.

`aggregate()` returns a *data frame* containing the names of the groups (`factor` confls).

```
> unique(mtcars$cyl)  # cyl has three unique values
> # Split mpg column based on number of cylinders
> split(mtcars$mpg, mtcars$cyl)
> # Split mtcars data frame based on number of cylinders
> split_cars <- split(mtcars, mtcars$cyl)
> str(split_cars, max.confl=1)
> names(split_cars)
> # Aggregate the mean mpg over split mtcars data frame
> sapply(split_cars, function(x) mean(x$mpg))
> # Or: split mpg column and aggregate the mean
> sapply(split(mtcars$mpg, mtcars$cyl), mean)
> # Same but using with()
> with(mtcars, sapply(split(mpg, cyl), mean))
> # Or: aggregate() using formula syntax
> aggregate(x=(mpg ~ cyl), data=mtcars, FUN=mean)
> # Or: aggregate() using data frame syntax
> aggregate(x=mtcars$mpg, by=list(cyl=mtcars$cyl), FUN=mean)
> # Or: using name for mpg
> aggregate(x=list(mpg=mtcars$mpg), by=list(cyl=mtcars$cyl), FUN=mea
> # Aggregate() all columns
> aggregate(x=mtcars, by=list(cyl=mtcars$cyl), FUN=mean)
```

The tapply() Functional

The functional tapply() is a specialized version of the apply() functional, that applies a function to elements of a *jagged array*.

A *jagged array* is a list consisting of vectors or matrices of different lengths.

tapply() accepts a vector of values "X", a factor "INDEX", and a function "FUN".

tapply() first groups the elements of "X" according to the factor "INDEX", transforming it into a *jagged array*, and then applies "FUN" to each element of the *jagged array*.

tapply() applies a function to sub-vectors aggregated using a factor, and performs *data pivoting* in a single function call.

The by() function is a wrapper for tapply().

The with() function evaluates an expression in an environment constructed from the data.

```
> # Mean mpg for each cylinder group
> tapply(X=mtcars$mpg, INDEX=mtcars$cyl, FUN=mean)
> # using with() environment
> with(mtcars, tapply(X=mpg, INDEX=cyl, FUN=mean))
> # Function sapply() instead of tapply()
> with(mtcars, sapply(sort(unique(cyl)), function(x) {
+  structure(mean(mpg[x==cyl]), names=x)
+      }))  # end with
> # Function by() instead of tapply()
> with(mtcars, by(data=mpg, INDICES=cyl, FUN=mean))
```

Data Pivoting Returning a Matrix

Sometimes *data pivoting* returns a list of vectors.

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

The function `do.call()` executes a function call using a function name and a list of arguments.

`do.call()` passes the list elements individually, instead of passing the whole list as one argument: `do.call(fun, list)= fun(list[[1]], list[[2]], ...)`

```
> # Get several mpg stats for each cylinder group
> data_cars <- sapply(split_cars,
+         function(x) {
+           c(mean=mean(x$mpg), max=max(x$mpg), min=min(x$mpg))
+         }  # end anonymous function
+         )  # end sapply
> data_cars  # sapply() produces a matrix
> data_cars <- lapply(split_cars,  # Now same using lapply
+         function(x) {
+           c(mean=mean(x$mpg), max=max(x$mpg), min=min(x$mpg))
+         }  # end anonymous function
+         )  # end sapply
> is.list(data_cars)  # lapply produces a list
> # do.call flattens list into a matrix
> do.call(cbind, data_cars)
```

Data Pivoting of Panel Data

The *data frame* `panel_data` contains fundamental financial data for *S&P500* stocks.

The `Industry` column has 22 unique elements, while the `Sector` column has 10 unique elements.

Each `Industry` belongs to a single `Sector`, but each `Sector` may have several `Industries` that belong to it.

The functional `aggregate()` allows aggregating over the `Industry` column, by perforing *data pivoting*.

```
> # Download CRSPpanel.txt from the NYU share drive
> # Read the file using read.table() with header and sep arguments
> paneld <- read.table(file="/Users/jerzy/Develop/lecture_slides/dat
+                      header=TRUE, sep="\t")
> paneld[1:5, 1:5]
> attach(paneld)
> # Split paneld based on Industry column
> panelds <- split(paneld, paneld$Industry)
> # Number of companies in each Industry
> sapply(panelds, NROW)
> # Number of Sectors that each Industry belongs to
> sapply(panelds, function(x) {
+   NROW(unique(x$Sector))
+ })  # end sapply
> # Or
> aggregate(x=(Sector ~ Industry),
+   data=paneld, FUN=function(x) NROW(unique(x)))
> # Industries and the Sector to which they belong
> aggregate(x=(Sector ~ Industry), data=paneld, FUN=unique)
> # Or
> aggregate(x=Sector, by=list(Industry), FUN=unique)
> # Or
> sapply(unique(Industry), function(x) {
+   Sector[match(x, Industry)]
+ })  # end sapply
```

Data Pivoting Returning a *Jagged Array*

A *jagged array* is a list consisting of vectors or matrices of different lengths.

The functional `aggregate()` returns a *data frame*, so it's output must be coerced if the *data pivoting* attempts to return a *jagged array*.

The functional `tapply()` returns an array, so it's output must be coerced if the *data pivoting* attempts to return a *jagged array*.

`tapply()` accepts a vector of values `"X"`, a factor `"INDEX"`, and a function `"FUN"`.

`tapply()` first groups the elements of `"X"` according to the factor `"INDEX"`, transforming it into a *jagged array*, and then applies `"FUN"` to each element of the *jagged array*.

`tapply()` applies a function to sub-vectors aggregated using a factor, and performs *data pivoting* in a single function call.

```
> # Split paneld based on Sector column
> panelds <- split(paneld, paneld$Sector)
> # Number of companies in each Sector
> sapply(panelds, NROW)
> # Industries belonging to each Sector (jagged array)
> secind <- sapply(panelds, function(x) unique(x$Industry))
> # Or use aggregate() (returns a data frame)
> secind2 <- aggregate(x=(Industry ~ Sector),
+   data=paneld, FUN=function(x) unique(x))
> # Or use aggregate() with "by" argument
> secind2 <- aggregate(x=Industry, by=list(Sector),
+     FUN=function(x) as.vector(unique(x)))
> # Coerce secind2 into a jagged array
> namesv <- secind2[, 1]
> secind2 <- secind2[, 2]
> names(secind2) <- namesv
> all.equal(secind2, secind)
> # Or use tapply() (returns an array)
> secind2 <- tapply(X=Industry, INDEX=Sector, FUN=unique)
> # Coerce secind2 into a jagged array
> secind2 <- drop(as.matrix(secind2))
> all.equal(secind2, secind)
```

Data Pivoting Over Multiple Columns

Data pivoting over multiple columns can be performed by splitting the *data frame* and then performing an sapply() loop using an anonymous function.

Splitting the *data frame* allows aggregations over multiple columns.

An anonymous function allows applying different aggregations on the same column.

```
> # Average ROE in each Industry
> sapply(split(ROE, Industry), mean)
> # Average, min, and max ROE in each Industry
> t(sapply(split(ROE, Industry), FUN=function(x)
+   c(mean=mean(x), max=max(x), min=min(x))))
> # Split paneld based on Industry column
> panelds <- split(paneld, paneld$Industry)
> # Average ROE and EPS in each Industry
> t(sapply(panelds, FUN=function(x)
+   c(mean_roe=mean(x$ROE),
+     mean_eps=mean(x$EPS.EXCLUDE.EI))))
> # Or: split paneld based on Industry column
> panelds <- split(paneld[, c("ROE", "EPS.EXCLUDE.EI")],
+   paneld$Industry)
> # Average ROE and EPS in each Industry
> t(sapply(panelds, FUN=function(x) sapply(x, mean)))
> # Average ROE and EPS using aggregate()
> aggregate(x=paneld[, c("ROE", "EPS.EXCLUDE.EI")],
+   by=list(paneld$Industry), FUN=mean)
```

draft: Compare `for()` Loops With *split-apply-combine* Procedure

from: Wickham Split Apply Combine for Data Analysis

The *split-apply-combine* procedure allows for analyzing *data frames*, by spliting an analysis into smaller steps, and then combining them together.

in data preparation, for performing group-wise ranking, standardization, or normalization.

creating summaries (aggregations) of data by groups.

During modeling, for fitting separate models to individual panels of data.

The *split-apply-combine* procedure is similar to the *map-reduce* procedure for processing large data.

also similar to *pivot tables* in *Excel*.

The `split()`, `apply()` and `combine` Procedure

The `lapply()` and `sapply()` functions are specialized versions

```
> library(plyr)
> one <- ozone[1, 1, ]
> month <- ordered(rep(1:12, length72))
> model <- rlm(one ~ month - 1)
> deseas <- resid(model)
> deseasf <- function(value) rlm(value ~ month - 1)
>
> # For loops
> models <- as.list(rep(NA, 24 * 24))
> dim(models) <- c(24, 24)
> deseas <- array(NA, c(24, 24, 72))
> dimnames(deseas) <- dimnames(ozone)
> for (i in seq_len(24)) {
+ for (j in seq_len(24)) {
+ mod <- deseasf(ozone[i, j, ])
+ models[[i, j]] <- mod
+ deseas[i, j, ] <- resid(mod)
+ }
+ }
>
> # Apply functions
> models <- apply(ozone, 1:2, deseasf)
> resids_list <- lapply(models, resid)
> resids <- unlist(resids_list)
> dim(resids) <- c(72, 24, 24)
> deseas <- aperm(resids, c(2, 3, 1))
> dimnames(deseas) <- dimnames(ozone)
>
```

draft: Examples of *split-apply-combine* Procedure

from:
http://4dpiecharts.com/2011/12/1
quick-primer-on-split-apply-combine-problems/

we have a *data frame* with one column containing the values to calculate a statistic for and another column containing the group to which that value belongs

```
> # InsectSprays dataset
> head(InsectSprays)
>
> # Split the count column by the spray column.
> count_by_spray <- with(InsectSprays, split(count, spray))
>
> # Next apply the statistic to each element of the list. Lets use the mean here.
> mean_by_spray <- lapply(count_by_spray, mean)
>
> # Finally combine the list as a vector
> unlist(mean_by_spray)
>
> # or in one line
> sapply(count_by_spray, mean)
>
> # Can also use the functions tapply(), aggregate() and by():
> with(InsectSprays, tapply(count, spray, mean))
> with(InsectSprays, by(count, spray, mean))
> aggregate(count ~ spray, InsectSprays, mean)
```

Exception Conditions: Errors and Warnings

Exception conditions are `R` objects containing information about *errors* or *warnings* produced while evaluating expressions.

The function `warning()` produces a *warning* condition, but doesn't halt function execution, and returns its message to the warning handler.

The function `stop()` produces an *error* condition, halts function execution, and returns its message to the error handler.

The handling of *warning* conditions depends on the value of `options("warn")`:

- *negative* then warnings are ignored,
- *zero* then warnings are stored and printed after the top-confl function has completed,
- *one* - warnings are printed as they occur,
- *two* or larger - warnings are turned into errors,

The function `suppressWarnings()` evaluates its expressions and ignores all warnings.

```
> # ?options  # Get info on global options
> getOption("warn")  # Global option for "warn"
> options("warn")  # Global option for "warn"
> getOption("error")  # Global option for "error"
> sqrt_safe <- function(input) {
+ # Returns its argument
+   if (input<0) {
+     warning("sqrt_safe: input is negative")
+     NULL  # Return NULL for negative argument
+   } else {
+     sqrt(input)
+   }  # end if
+ }  # end sqrt_safe
> sqrt_safe(5)
> sqrt_safe(-1)
> options(warn=-1)
> sqrt_safe(-1)
> options(warn=0)
> sqrt_safe()
> options(warn=1)
> sqrt_safe()
> options(warn=3)
> sqrt_safe()
```

Validating Function Arguments

Argument validation consists of first determining if any arguments are *missing*, and then determining if the arguments are of the correct *type*.

An argument is *missing* when the formal argument is not bound to an actual value in the function call.

The function `missing()` returns `TRUE` if an argument is missing, and `FALSE` otherwise.

Missing arguments can be detected by:
- assigning a `NULL` default value to formal arguments and then calling `is.null()` on them,
- calling the function `missing()` on the arguments.

The argument *type* can be validated using functions such as `is.numeric()`, `is.character()`, etc.

The function `return()` returns its argument and terminates futher function execution.

```
> # Function valido validates its arguments
> valido <- function(input=NULL) {
+ # Check if argument is valid and return double
+   if (is.null(input)) {
+     return("valido: input is missing")
+   } else if (is.numeric(input)) {
+     2*input
+   } else cat("valido: input not numeric")
+ }  # end valido
> valido(3)
> valido("a")
> valido()
> # valido validates arguments using missing()
> valido <- function(input) {
+ # Check if argument is valid and return double
+   if (missing(input)) {
+     return("valido: input is missing")
+   } else if (is.numeric(input)) {
+     2*input
+   } else cat("valido: input is not numeric")
+ }  # end valido
> valido(3)
> valido("a")
> valido()
```

Validating Assertions Inside Functions

If assertions about variables inside a function are `FALSE`, then `stop()` can be called to halt its execution.

Calling `stop()` is preferable to calling `return()`, or inserting `cat()` statements into the code.

Using `stop()` inside a function allows calling the function `traceback()`, if an error was produced.

The function `traceback()` prints the call stack, showing the function that produced the *error* condition.

`cat()` statements inside the function body provide information about the state of its variables.

```
> # valido() validates its arguments and assertions
> valido <- function(input) {
+ # Check if argument is valid and return double
+   if (missing(input)) {
+     stop("valido: input is missing")
+   } else if (!is.numeric(input)) {
+     cat("input =", input, "\n")
+     stop("valido: input is not numeric")
+   } else 2*input
+ }  # end valido
> valido(3)
> valido("a")
> valido()
> # Print the call stack
> traceback()
```

Validating Assertions Using stopifnot()

R provides robust validation and debugging tools through *type* validation functions, and functions missing(), stop(), and stopifnot().

If the argument to function stopifnot() is FALSE, then it produces an *error* condition, and halts function execution.

stopifnot() is a convenience wrapper for stop(), and eliminates the need to use if () statements.

stopifnot() is often used to check the validity of function arguments.

stopifnot() can be inserted anywhere in the function body in order to check assertions about its variables.

```
> valido <- function(input) {
+ # Check argument using long form '&&' operator
+   stopifnot(!missing(input) && is.numeric(input))
+   2*input
+ }  # end valido
> valido(3)
> valido()
> valido("a")
> valido <- function(input) {
+ # Check argument using logical '&' operator
+   stopifnot(!missing(input) & is.numeric(input))
+   2*input
+ }  # end valido
> valido()
> valido("a")
```

Validating Function Arguments and Assertions

The functions `stop()` and `stopifnot()` halt function execution and produce *error* conditions if certain assertions are `FALSE`.

The *type* validation functions, such as `is.numeric()`, `is.na()`, etc., and `missing()`, allow for validation of arguments and variables inside functions.

`cat()` statements can provide information about the state of variables inside a function.

`cat()` statements don't return values, so they provide information even when a function produces an `error`.

```
> # sumtwo() returns the sum of its two arguments
> sumtwo <- function(input1, input2) {  # Even more robust
+ # Check if at least one argument is not missing
+   stopifnot(!missing(input1) && !missing(input2))
+ # Check if arguments are valid and return sum
+   if (is.numeric(input1) && is.numeric(input2)) {
+     input1 + input2  # Both valid
+   } else if (is.numeric(input1)) {
+     cat("input2 is not numeric\n")
+     input1  # input1 is valid
+   } else if (is.numeric(input2)) {
+     cat("input1 is not numeric\n")
+     input2  # input2 is valid
+   } else {
+     stop("none of the arguments are numeric")
+   }
+ }  # end sumtwo
> sumtwo(1, 2)
> sumtwo(5, 'a')
> sumtwo('a', 5)
> sumtwo('a', 'b')
> sumtwo()
```

The R Debugger Facility

The function `debug()` flags a function for future debugging, but doesn't invoke the debugger.

After a function is flagged for debugging with the call `"debug(myfun)"`, then the function call `"myfun()"` automatically invokes the debugger (browser).

When the debugger is first invoked, it prints the function code to the console, and produces a *browser* prompt: `"Browse[2]>"`.

Once inside the debugger, the user can execute the function code one command at a time by pressing the *Enter* key.

The user can examine the function arguments and variables with standard R commands, and can also change the values of objects or create new ones.

The command `"c"` executes the remainder of the function code without pausing.

The command `"Q"` exits the debugger (browser).

The call `"undebug(myfun)"` at the R prompt unflags the function for debugging.

```
> # Flag "valido" for debugging
> debug(valido)
> # Calling "valido" starts debugger
> valido(3)
> # unflag "valido" for debugging
> undebug(valido)
```

Debugging Using browser()

As an alternative to flagging a function for debugging, the user can insert the function `browser()` into the function body.

`browser()` pauses the execution of a function and invokes the debugger (browser) at the point where `browser()` was called.

Once inside the debugger, the user can execute all the same browser commands as when using `debug()`.

`browser()` is usually inserted just before the command that is suspected of producing an *error* condition.

Another alternative to flagging a function for debugging, or inserting `browser()` calls, is setting the `"error"` option equal to `"recover"`.

Setting the `"error"` option equal to `"recover"` automatically invokes the debugger when an *error* condition is produced.

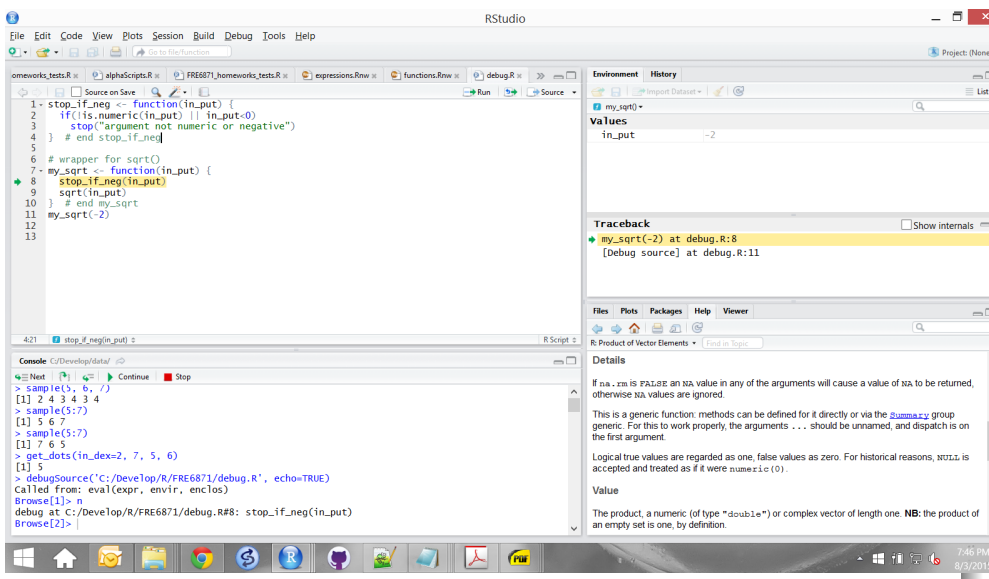
```
> valido <- function(input) {
+   browser()  # Pause and invoke debugger
+ # Check argument using long form '&&' operator
+   stopifnot(!missing(input) && is.numeric(input))
+   2*input
+ }  # end valido
> valido()  # Invokes debugger
> options("error")  # Show default NULL "error" option
> options(error=recover)  # Set "error" option to "recover"
> options(error=NULL)  # Set back to default "error" option
```

Using the Debugger in *RStudio*

RStudio has several built-in debugging facilities that complement those already installed in `R`:

- toggling breakpoints, instead of inserting `browser()` commands,
- stepping into functions,
- environment pane with environment stack, instead of calling `ls()`,
- traceback pane, instead of calling `traceback()`,

RStudio provides an online debugging tutorial:
https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio



Handling Exception Conditions

The function `tryCatch()` executes functions and expressions, and handles any *exception conditions* produced when they are evaluated.

`tryCatch()` first evaluates its `"expression"` argument.

If no error or warning `condition` is produced then `tryCatch()` just returns the value of the expression.

If an `exception condition` is produced then `tryCatch()` invokes error and warning *handlers* and executes other expressions to provide information about the `exception condition`.

If a *handler* is provided to `tryCatch()` then the error is captured by the *handler*, instead of being broadcast to the console.

At the end, `tryCatch()` evaluates the expression provided to the `finally` argument.

```
> str(tryCatch)  # Get arguments of tryCatch()
> tryCatch(  # Without error handler
+   {  # Evaluate expressions
+     numv <- 101  # Assign
+     stop('my error')  # Produce error
+   },
+   finally=print(paste("numv=", numv))
+ )  # end tryCatch
>
> tryCatch(  # With error handler
+   {  # Evaluate expressions
+     numv <- 101  # Assign
+     stop('my error')  # Produce error
+   },
+   # Error handler captures error condition
+   error=function(error_cond) {
+     print(paste("error handler: ", error_cond))
+   },  # end error handler
+   # Warning handler captures warning condition
+   warning=function(warning_cond) {
+     print(paste("warning handler: ", warning_cond))
+   },  # end warning handler
+   finally=print(paste("numv=", numv))
+ )  # end tryCatch
```

Error Conditions in Loops

If an *error* occurs in an `apply()` loop, then the loop exits without returning any result.

`apply()` collects the values returned by the function supplied to its `FUN` argument, and returns them only after the loop is finished.

If one of the function calls produces an error, then the loop is interrupted and `apply()` exits without returning any result.

The function `tryCatch()` captures errors, allowing loops to continue after the error `condition`.

```
> # Apply loop without tryCatch
> apply(matrix(1:5), 1, function(numv) {  # Anonymous function
+     stopifnot(!(numv = 3))  # Check for error
+     # Broadcast message to console
+     cat("(cat) numv =", numv, "\n")
+     # Return a value
+     paste("(return) numv =", numv)
+   }  # end anonymous function
+ )  # end apply
```

Exception Handling in Loops

If the body of the function supplied to the `FUN` argument is wrapped in `tryCatch()`, then the loop can finish without interruption and return its results.

The messages produced by *errors* and *warnings* can be caught by *handlers* (functions) that are supplied to `tryCatch()`.

The *error* and *warning* messages are bound (passed) to the formal arguments of the *handler* functions that are supplied to `tryCatch()`.

`tryCatch()` always evaluates the expression provided to the `finally` argument, even after an *error* occurs.

```
> # Apply loop with tryCatch
> apply(as.matrix(1:5), 1, function(numv) {  # Anonymous function
+     tryCatch(  # With error handler
+ {  # Body
+   stopifnot(numv != 3)  # Check for error
+   # Broadcast message to console
+   cat("(cat) numv =", numv, "\t")
+   # Return a value
+   paste("(return) numv =", numv)
+ },
+ # Error handler captures error condition
+ error=function(error_cond)
+   paste("handler: ", error_cond),
+ finally=print(paste("(finally) numv =", numv))
+     )  # end tryCatch
+   }  # end anonymous function
+ )  # end apply
```