Merged Functions

Samantha Alger, Lauren Ash, Lauren Ashlock, Alex Burnham, Peter Clark, Melanie Kazenal, Erin Keller, Alex Looi, April Makukhov, Emily Makucki, Mathias Nevins, Morgan Southgate

March 8, 2017

&

Alex Burnham

Description: This character is a logical operator and acts on logical and numeric (numeric-like) vectors. single “&” indicates each value within the vector is evaluated and in && only the first value is evaluated.

Arguments:

- x = vector that is to be operated on
- y = object with operations within

Example:

```
# create vector of values 1 through 10:
x <- c(1:10)
# create vector of values 11 through 20:
y <- x>5

# single and symbol
y & x
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
# double and symbol
y && x
```

```
## [1] FALSE
```

Explanation of Example: In this case, the opparator puts out a string of logicals indcating where the values of x are greater than 5 and the second example does this only to the first value in the vector.

<-

Alex Burnham

Description: This operator, the assingment opperator is the most powerful and usefull character in the R language. It is used to store information into objects creating various data structures. It can be thought of as a unidirectional equals sign.

Arguments:

- x = object
- value = your data

Example:

```
# store 10 random uniform numbers in a vector using <-
x <- runif(10)
print(x)
```

```
##  [1] 0.8321808 0.1876172 0.6065788 0.2378810 0.2518657 0.3306815 0.4808096
##  [8] 0.9373472 0.7256194 0.4660656
```

Explanation of Example: In this example the vector x was created by using the assingment operator to store 10 randum uniform numbers in x

abs

April D. Makukhov

This function computes the absolute value of a numeric or complex vector. Recall from mathematics that the absolute value of a number (or variable representing numbers, such as x) is the positive value of that number or variable, so using the abs function would cause your value(s) to all be positive (unless you were to use any other arithmetic operators that may change that).

```
abs(2)
```

```
## [1] 2
```

```
abs(-2) # these outputs should be the same, because the absolute, or 'positive', value for digits
 2 and -2 is "2".
```

```
## [1] 2
```

```
x <- c(2, 3, -8, -9.7, 4)
abs(x) # here, you will notice that the outputs for all of these value, even for this more complex
 vector, are positive
```

```
## [1] 2.0 3.0 8.0 9.7 4.0
```

acos

Lauren Ashlock

The acos function calculates the arc-cosine for a given argument, x. The arccosine is the inverse cosine function of x, when -1<x<1.

```
#acos function

z<- c(.2,.3,.4,.5)
acos(z)
```

```
## [1] 1.369438 1.266104 1.159279 1.047198
```

all

Matthias Nevins

You can use the ‘all’ function to see if a given set of logical vectors contains all true values.

- USE: all(…,na.rm = FALSE)
- Arguments -… zero or more logical vectors
- na.rm A logical argument where if TRUE, NA values are removed before computing the results.

```
# You can assess a set of logical vectors using the 'all' function
# lets begin by defining a variable
x <- 1:10
# Now we can assess the variable using 'all()'
all(x>11)
```

```
## [1] FALSE
```

```
# This returns FALSE because the variable x is a set of numbers from 1:10
```

all.equal

Samantha Alger

all.equal(target,current) will check if two objects are equal, or nearly equal.

- target-An R object
- current- A second R object to compare to target (the first argument)

Output will give the difference betwen the objects or will return TRUE if the values are sufficiently close enough

```
all.equal(pi, 3.14)
```

```
## [1] "Mean relative difference: 0.0005069574"
```

```
all.equal(pi, 3.1415)
```

```
## [1] "Mean relative difference: 2.949255e-05"
```

```
all.equal(pi, 3.141592)
```

```
## [1] "Mean relative difference: 2.080441e-07"
```

```
all.equal(pi, 3.14159265)
```

```
## [1] TRUE
```

`any()`

Alex Burnham

Description: Given a set of logical vectors, is at least one of the values true?

Arguments:

- 1 or more logical vectors
- na.rm = if TRUE, na values are removed

Example:

```
# two vectors of 10 values:
set.seed(100)
x <- rnorm(10)
y <- rnorm(10)
# logical statements (y greater than x - TRUE or FALSE) or the opposite
z <- y > x
t <- y < x

# one logical vector
any(z)
```

```
## [1] TRUE
```

```
# evaluating 2 logical vectors
any(z,t)
```

```
## [1] TRUE
```

Explanation of Example: Returns a TRUE because at least on of the values is in fact TRUE. Second example also returns TRUE because both vectors have at least one TRUE value contained within.

`as.logical`

Peter Clark

One of the logical vectors, **as.logical** creates or tests for objects of type "logical". It attempts to coerce its argument to be of logical type, returning a *TRUE* or *FALSE*. Values <0 are *FALSE* and those >1 are *TRUE*. **as.logical** strips attributes including names. Character strings *c("T", "TRUE", "True", "true")* are regarded as true, *c("F", "FALSE", "False", "false")* as false, and all others as *NA*.

Usage

as.logical(x, …) where *x* is the object to be coerced or tested

```
a <- 3
x <- as.logical(a > 2) # coereces the variable a into a logical (TRUE, FALSE) if the argument is m
et
print(x)          # let's see the output
```

```
## [1] TRUE
```

```
if(x) TRUE    # returns a logical if the parements in the if statement are true
```

```
## [1] TRUE
```

```
x <- 0:6
as.logical(x) # coerce each bit in sequence to logical
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

as.matrix()

Alex Burnham

Description: This function allows you to convert another data structure into matrix form. **Arguments:**

- x = dataframe or another R object
- dimnames = (character string) and gives names to the dimensions of the matrix

Example:

```
# make a dataframe:
dataframe <- data.frame(cbind(c("blue", "red", "green"), c(1:3)))
print(dataframe)
```

```
##      X1 X2
## 1  blue  1
## 2   red  2
## 3 green  3
```

```
# convert data frame to a matrix:
matrix <- as.matrix(x=dataframe)
str(matrix)
```

```
##  chr [1:3, 1:2] "blue" "red" "green" "1" "2" "3"
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "X1" "X2"
```

Explanation of Example: The dataframe, which had two variable types (character and numeric), was converted into a dataframe of the same dimensions. Numeric variables were coerced into characters because a matrix must be of all the same variable type and numeric variables get coerced into characters as part of R's hierarchical structure.

`as.character()`

Alexander Looi

This function will convert numerics, ints, factors, date, etc. data types to type characer. This function can take in a vector of any type and will convert the vectors to a vector of characters. The same can be done with matrices and data frames. However, the result is not what you may expect. as.character will always return a vector of characters. You will lose the "data structure". Lastly, becarefuly when you give as.character a vector of 'factor' type. as.character will return the numeric 'factor' value and not the character version. Additionally, e careful with the kinds of objects you give, object types other than the ones listed above may not be coercible to a string.

```
# numerics can be turned to characters, but they lose their "numeric" properties
q = 1:5
q_c = as.character(q)
class(q_c)
```

```
## [1] "character"
```

```
# matrices can be converted to characters, noe that the entire matrix will converted
# to a character type. Also becareful, since the matrix will lose it's structure
A = matrix(c(1:6), 2, 3)
A_c = as.character(A)
A_c
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

```
# Will also work on data frames, but it will entire convert columns into a single
# character string and return a vector of length number of columns. Also notice
# the 'a', 'b', 'c' variables have been converted to 1, 2, 3's.
a = 1:5
b = letters[1:3]
y = expand.grid(a,b, stringsAsFactors = T)
as.character(y)
```

```
## [1] "c(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)"
## [2] "c(1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3)"
```

```
# better to convert a column at a time
as.character(y$Var1)
```

```
##  [1] "1" "2" "3" "4" "5" "1" "2" "3" "4" "5" "1" "2" "3" "4" "5"
```

```
# notice the 'a', 'b', 'c' variables have been preserved
# eventhough stringsAsFactors = T
as.character(y$Var2)
```

```
##  [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "c" "c" "c" "c" "c"
```

as.data.frame

Erin L. Keller

as.data.frame will report whether a given input value is a data frame. This is important as some functions require data to be in a data frame format (which is similar to a matrix but is distinct from a matrix). The input value can be any element of interest and the output value will be a logical vector, where “TRUE” indicates that the element of interest is in data frame format while “FALSE” indicates that the element of interest is not in data frame format. To coerce data into a data frame, use the as.data.frame function.

```
matrix<-matrix(1:10,2,5)
is.data.frame(matrix)
```

```
## [1] FALSE
```

```
matrix2<-as.data.frame(matrix)
is.data.frame(matrix2)
```

```
## [1] TRUE
```

asin

Melanie R. Kazenel

The `asin` function computes the arc-sine of a numeric or complex vector, which is the input. The input should be in radians, not degrees. The output is the arc-sine value of each number in the vector.

```
# Compute the arc-sine of a single number
asin(1)
```

```
## [1] 1.570796
```

```
# Compute the arc-sine of each number in a vector
z <- c(0.5,0.75,1) # create a vector
asin(z) # compute the arcsine of each element
```

```
## [1] 0.5235988 0.8480621 1.5707963
```

as.numeric

Emily Mikucki

The function as.numeric creates/coerces objects of type ‘numeric.’ The input ‘x’ should be the object to be coerced/tested and other arguments are not usually used. For example, an object that includes a character will coerce the character to the numeric ‘NA’ (and give a warning informing of the coercion).

Be warned, if x is a factor, as.numeric will return the underlying integer representation, but it will often be meaningless because they most likely won’t correspond to factor levels.

```
## If a character is in the list, it will return NA and a warning
as.numeric(c("-.1"," 2.7 ","B"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] -0.1  2.7   NA
```

```
## Input a matrix and you will obtain a numerical object
mat<-matrix(1:12, 4, 3)
print(mat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
class(mat) # you can see the class of the input is matrix
```

```
## [1] "matrix"
```

```
num<-as.numeric(mat)
print(num)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
class(num) # the class of the output is numeric
```

```
## [1] "numeric"
```

`assign`

Samantha Alger

assign() assigns a value to a name assign(x, value, pos=, envir=…) * x - variable name * value - will be assigned to the variable * pos - position to do assignment * envir - the environment to use

```
assign("z", 5)
z
```

```
## [1] 5
```

atan

Morgan W. Southgate

The atan function is the inverse of the tan function, and so returns the y value for which x= tan(y).

atan(x)

Input:
- x: numeric or complex vector signifying angle theta.

Output:
- the y value for which x = tan(y), in terms of radians.

```
# Calculates the value of y for which 5 = tan(y)
atan(5)
```

```
## [1] 1.373401
```

atan2

Samantha Alger

atan2(x,y) is a trigonometric function that computes the arc-tangent of two arguments. It will return the angle between the x-axis and the vector (x,y)

```
atan2(3,4)
```

```
## [1] 0.6435011
```

c

Morgan W. Southgate

The c function concatenates values into a vector or a list. All arguments are coerced to a common type in the order logical -> integer -> double -> character.Logical arguments are coerced to integer arguments using the system TRUE = 1, FALSE = 0.

c(….)

Inputs:
- ….: the objects to be concatenated

Output:
- an atomic vector or list of concatenated objects

```
# Concatenate the numeric variables 1 & 2 and the logical variable TRUE
c1 <- c(1,2,TRUE)

# Check the structure of c1 - as two different types of variables were concatened, the logical var
iable should have been coerced into a numeric variable of the value 1.
str(c1)
```

```
##  num [1:3] 1 2 1
```

c()

Peter Clark

c() is a function which combines (or more specificially, concatenate) its arguments and returns a vector or list (a one dimensional array). When you use **c()** to create a vector, what you are actually doing is combining together a series of 1-length vectors. You may also combine vectors and lists.

Usage

c(…) where … are objects to be concatenated

```
# combining two variables
k <- (1:5)
b <- (10:20) # k and b pruduce a sequence function for two different sets of numbers
d <- c(k,b) # c combines those two sequences together and returns them as one list
print(d)
```

```
##  [1]  1  2  3  4  5 10 11 12 13 14 15 16 17 18 19 20
```

```
# combining more
e<-c(c(1, 2), c(d))
print(e)
```

```
##  [1]  1  2  1  2  3  4  5 10 11 12 13 14 15 16 17 18 19 20
```

```
# combining a vector and list
x = 1:10  # create a vector variable
is.vector(x) # is it a vector? Yes indeed
```

```
## [1] TRUE
```

```
y = 3*x+rnorm(length(x)) # create a new vector
is.vector(y) # still a vector
```

```
## [1] TRUE
```

```
z = lm(y ~ x) # running these as a linear model is not a vector, but is a list
is.list(z) # the proof is in the pudding
```

```
## [1] TRUE
```

```
combine = c(x, z) # combine a vector and list creates a table
print(combine)
```

```
## [[1]]
## [1] 1
## 
## [[2]]
## [1] 2
## 
## [[3]]
## [1] 3
## 
## [[4]]
## [1] 4
## 
## [[5]]
## [1] 5
## 
## [[6]]
## [1] 6
## 
## [[7]]
## [1] 7
## 
## [[8]]
## [1] 8
## 
## [[9]]
## [1] 9
## 
## [[10]]
## [1] 10
## 
## $coefficients
## (Intercept)           x 
##   0.2557266   2.9300238 
## 
## $residuals
##          1          2          3          4          5          6 
## -0.6238404  0.6482863  0.2161632  0.7975827 -0.7202249 -0.2743201 
##          7          8          9         10 
## -0.4861149  0.5350274 -0.7836705  0.6911112 
## 
## $effects
##  (Intercept)            x                                                
## -51.76919737  26.61326293   0.26758120   0.91839548  -0.53001726 
##                                                                   
##  -0.01471772  -0.15711772   0.93341934  -0.31588367   1.22829277 
## 
## $rank
## [1] 2
## 
## $fitted.values
##         1         2         3         4         5         6         7 
##  3.185750  6.115774  9.045798 11.975822 14.905846 17.835870 20.765893 
##         8         9        10 
## 23.695917 26.625941 29.555965
##
## $assign
## [1] 0 1
##
## $qr
## $qr
##    (Intercept)            x
## 1   -3.1622777 -17.39252713
## 2    0.3162278   9.08295106
## 3    0.3162278   0.15621147
## 4    0.3162278   0.04611510
## 5    0.3162278  -0.06398128
## 6    0.3162278  -0.17407766
## 7    0.3162278  -0.28417403
## 8    0.3162278  -0.39427041
## 9    0.3162278  -0.50436679
## 10   0.3162278  -0.61446316
## attr(,"assign")
## [1] 0 1
##
## $qraux
## [1] 1.316228 1.266308
##
## $pivot
## [1] 1 2
##
## $tol
## [1] 1e-07
##
## $rank
## [1] 2
##
## attr(,"class")
## [1] "qr"
##
## $df.residual
## [1] 8
##
## $xlevels
## named list()
##
## $call
## lm(formula = y ~ x)
##
## $terms
## y ~ x
## attr(,"variables")
## list(y, x)
## attr(,"factors")
##   x
## y 0
## x 1
## attr(,"term.labels")
## [1] "x"
## attr(,"order")
## [1] 1
## attr(,"intercept")
## [1] 1
## attr(,"response")
## [1] 1
## attr(,".Environment")
## <environment: R_GlobalEnv>
## attr(,"predvars")
## list(y, x)
## attr(,"dataClasses")
##         y         x 
## "numeric" "numeric" 
## 
## $model
##            y  x
## 1   2.561910  1
## 2   6.764061  2
## 3   9.261961  3
## 4  12.773405  4
## 5  14.185621  5
## 6  17.561549  6
## 7  20.279778  7
## 8  24.230945  8
## 9  25.842271  9
## 10 30.247076 10
```

`capture.output()`

Alex Burnham

Description: Evaluates its arguments with the output being returned as a character string or sent to a file. Related to sink in the same way that with is related to attach.

Arguments:

- first argument is expression to be evaulated
- file = filename to store output in (or NULL)
- append = TRUE or FALSE (apend or overwrite)

Example:

```
# make a data frame:
dataframe <- data.frame(cbind(rep(c("blue", "red", "green"),3), c(1:9)))
dataframe$X1 <- as.factor(dataframe$X1)
dataframe$X2 <- as.numeric(dataframe$X2)
print(dataframe)
```

```
##      X1 X2
## 1  blue  1
## 2   red  2
## 3 green  3
## 4  blue  4
## 5   red  5
## 6 green  6
## 7  blue  7
## 8   red  8
## 9 green  9
```

```
# run an ANOVA model:
model <- aov(dataframe$X2~dataframe$X1)

# use capture to
capture.output(model, file = NULL)
```

```
##  [1] "Call:"
##  [2] "   aov(formula = dataframe$X2 ~ dataframe$X1)"
##  [3] ""
##  [4] "Terms:"
##  [5] "                dataframe$X1 Residuals"
##  [6] "Sum of Squares             6        54"
##  [7] "Deg. of Freedom            2         6"
##  [8] ""
##  [9] "Residual standard error: 3"
## [10] "Estimated effects may be unbalanced"
```

Explanation of Example: The output takes the summary output of the model and displays it and saves it

^

April D. Makukhov

This symbol is an operator used in mathematics to symbolize an exponent. It can be used on numeric or complex vectors or objects labeled as a numeric value.

```
10^2 # This calculates 10 to the 2nd power
```

```
## [1] 100
```

```
x=3
y <- x^5 # Here were are setting y to be represented as x to the 5th power, using the ^ symbol to
 set 5 as the exponential value
print(y)
```

```
## [1] 243
```

```
cat=12
dog=3
cat^dog # we can do this with characters as well as long as those characters are represented by a
 numeric value
```

```
## [1] 1728
```

cat

Samantha Alger

Print output to the screen or to a file. Use 'cat' instead of 'print' to print information from a function. cat(…, file "", sep="",append=FALSE)

- … -The information to be printed to the screen or saved file
- file- Specifies a file to be created (this is optional)
- sep -Specifies what separates the objects in…that are to be printed.
- append -If a file is specified, then indicate whether to append to the content in the existing file (the default is not to append, which means to overwrite the existing content).

```
x <- c(1:10)
print(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# Space separator
cat(x)
```

```
## 1 2 3 4 5 6 7 8 9 10
```

```
# Multi-space separator, and adds a period between each number
cat(x, sep=" . ")
```

```
## 1 . 2 . 3 . 4 . 5 . 6 . 7 . 8 . 9 . 10
```

```
# Separates each number by a tab
cat(x, sep="\t")
```

```
## 1    2   3   4   5   6   7   8   9   10
```

```
# Separates each number by a line break
cat(x, sep="\n")
```

```
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
## 10
```

```
# Use cat to index particular values
# and specify how those values will be printed using sep

# creating a data set (y) of 10 random atomic components of either red or blue
y <- sample(c("red", "blue"), 10, TRUE)
y
```

```
##  [1] "red"  "blue" "blue" "blue" "red"  "red"  "red"  "red"  "red"  "blue"
```

```
# use cat to concatenate and print the two atomic components x and y
cat(x, y)
```

```
## 1 2 3 4 5 6 7 8 9 10 red blue blue blue red red red red red blue
```

```
#use cat to specify which values to print (first value of each vector) and separate those two valu
es by a tab.
cat(x[1], y[1], sep="\t")
```

```
## 1    red
```

cbind

Emily Mikucki

The `cbind` function allows you to combine vectors, matrices or data frames by columns. In the default method, all the vectors and matrices must be atomic vectors or lists. If you have multiple matrix arguments, they must have the same number of columns to be able to use this function. If you have only vector arguments, then the number of columns in the result is equal to the length of the longest vector. Values in shorter arguments are recycled to achieve the length of the longest vector. When you have matrices and vectors, the number of columns is equal to that of the matrix and if the vector is shorter, then they will have their values recycled or subsetted to match the matrix. When using `cbind` , vectors with a length of 0 (or NULL) will be ignored. Names are reported from the `colnames` argument. When using `cbind` with a data frame, matrix columns will be split into data frame arguments. You must specify stringsAsFactors = FALSE or the character columns will be converted to factors.

```
m <- matrix(data=1:12, nrow = 4, ncol = 3) #Create a matrix just like in-class demo
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
m2 <- cbind(c(0.2, 0.4, 0.6, 0.8), m) #Create a new column BEFORE the rest of the data
m2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.2    1    5    9
## [2,]  0.4    2    6   10
## [3,]  0.6    3    7   11
## [4,]  0.8    4    8   12
```

```
m2 <- cbind(m, c(0.2, 0.4, 0.6, 0.8)) #Create a new column AFTER the rest of the data
m2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9  0.2
## [2,]    2    6   10  0.4
## [3,]    3    7   11  0.6
## [4,]    4    8   12  0.8
```

```
#This is a very simple example, so make sure your headers match the data appropriately when using
 cbind
```

ceiling()

Alex Burnham

Description: This function takes a vector (x) and returns a vector that rounds all values up to their nearest integer value. For instance 3.4 becomes 4 not 3.

Arguments:

- x = a vector

Example:

```
# create vector of values:
x <- c(0.4, 2.3, 9.4, 3.4, 4.2)

ceiling(x)
```

```
## [1]  1  3 10  4  5
```

Explanation of Example: All values got rounded up to their nearest integer values.

choose

Melanie R. Kazenel

The `choose` function can be used to calculate binomial coefficients. In other words, it can be used to calculate “n choose k” – the number of ways to to choose k elements from from a set of n elements, where n is a number and k is an integer. As arguments, the function takes a numeric vector for n and an integer vector for k. The function will round non-integer values of k to integers by default. When the vectors have a lenghth of one, the output is a single “n choose k” value. When the vectors contain multiple elements, the function will calculate “n choose k” for the pairs of elements in corresponding positions in the vectors.

```
# Calculate the number of ways to choose 2 elements from a set of 8 elements
choose(n=8,k=2)
```

```
## [1] 28
```

```
# Calculate the number of ways to choose 3 elements from a set of 4 elements
choose(n=4,k=3)
```

```
## [1] 4
```

```
# If vectors containing equal numbers of elements are used as input for n and k, the function will
 calculate "n choose k" for pairwise combinations of elements in corresponding positions in the ve
ctors.
choose(n=c(8,4),k=c(2,3))
```

```
## [1] 28  4
```

```
# In the example below, the vectors for n and k contain unequal numbers of elements. The value of
 k is used to calculate "n choose k" for both values in n.
choose(n=c(8,4),k=2)
```

```
## [1] 28  6
```

colnames()

Alex Burnham

Description: Retrieve or set the column names of a matrix-like object. The names are in a list and stored in the object using the assignment operator “<-”.

Arguments:

- x = a matrix-like object (matrix or dataframe) with at least two dimensions

- <- value is a list of names that is stored in the object "x"
- do.null = TRUE or FLASE (default is TRUE), if FALSE and names are NULL, names are created.

Example:

```
# make a dataframe:
dataframe <- data.frame(cbind(c("blue", "red", "green"), c(1:3)))
print(dataframe)
```

```
##      X1 X2
## 1  blue  1
## 2   red  2
## 3 green  3
```

```
# assign names to columns and print:
colnames(dataframe) <- c("color", "number")
print(dataframe)
```

```
##   color number
## 1  blue      1
## 2   red      2
## 3 green      3
```

Explanation of Example: The names "color" and "number" were added to the two columns of the data frame previously known by generic varable names x1 and x2.

combn()

Alex Burnham

Description: This function determines the combinations that can be generated from an atomic vector taken m number of ways and returns a matrix, array or list of these combinations.

Arguments:

- x = vector source for combinations
- m = number of elements to choose
- FUN = is a function that can be applied to matrices (by default its the identity matrix)
- simplify = TRUE or FALSE (by default TRUE) asking if it should be simplified into a matrix (or another array) or a list if FALSE

Example:

```
# create vector of values 1 through 10:
x <- c(1:10)

# combine 1 through 10 two ways and return a matrix:
combn(x=x, m = 2, simplify = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]    1    1    1    1    1    1    1    1    1     2     2     2     2
## [2,]    2    3    4    5    6    7    8    9   10     3     4     5     6
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
## [1,]     2     2     2     2     3     3     3     3     3     3     3
## [2,]     7     8     9    10     4     5     6     7     8     9    10
##      [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
## [1,]     4     4     4     4     4     4     5     5     5     5     5
## [2,]     5     6     7     8     9    10     6     7     8     9    10
##      [,36] [,37] [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45]
## [1,]     6     6     6     6     7     7     7     8     8     9
## [2,]     7     8     9    10     8     9    10     9    10    10
```

Explanation of Example: This example shows how the values 1 through 10 can be taken 2 ways in combination. There are 45 ways to combine 2 different values from this vector together without repeating combinations.

complete.cases

Erin L. Keller

The function complete.cases indicates what values in your data (vectors, matrix, data frames) are complete (do not have missing values, no NA). The input for this function needs to be a vector, matrix, or data frame and the output will be a logical vector with "TRUE" indicating that the value is complete (no missing data) while a "FALSE" indicates that the value is incomplete (missing data, NA). This function is useful for identifying missing data in data sets, although if the intent is to discard NA values, na.omit is a better function to use.

```
test<-c(0,1.5,NA,5,4.5,NA,3,3,NA)
complete.cases(test)
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

cor()

Alexander Looi

This function gives the user the correlation between two variables. It is usually most intuitive to pass specific vectors for x and y (which need to be the same length), but this function can receive data frames and matrices. If given either it will find the correlations between all combinations of columns. If there are na's in your data set be sure to set the na.rm to TRUE. Additionally, you can specify different 'methods' and 'use'. 'use' tells the function what method to use to calculate the correlation, and 'method' tells the function which kind of correlation coefficient to be calculated.

```
x = 1:20
y_1 = runif(20)
y_2 = 1:20

cor(x, y_1) # not correlated
```

```
## [1] -0.1141505
```

```
cor(x, y_2) # is correlated
```

```
## [1] 1
```

```
# can take in dataframes, note that it does all combinations and returns a matrix.
DF = data.frame(x,y_1,y_2)
cor(DF)
```

```
##              x        y_1        y_2
## x    1.0000000 -0.1141505  1.0000000
## y_1 -0.1141505  1.0000000 -0.1141505
## y_2  1.0000000 -0.1141505  1.0000000
```

cos

Peter Clark

One of the *Trig* functions, the **cos()** function computes the cosine value of numeric value. See Trig in help menu for a suite of other arguments Note: R always works with angles in radians, not in degrees.

Usage

cos(x) where *x* is a numeric value, array or vector

```
cos(120*pi/180) #  since R doesn't work in degrees, use pi/180 to calculate the cosine of an angle
 of 120 degrees
```

```
## [1] -0.5
```

```
x <- c(pi, pi/4, pi/3)
cos(x)
```

```
## [1] -1.0000000  0.7071068  0.5000000
```

count.fields()

Alex Burnham

Description: This functon counts how many fields are in a data on each line of file being loaded in by counting what is between the seperator in the file.

Arguments:

- file = file name
- sep = the type of seperation in the file (comma "," if .csv)
- skip = how many lines to skip before reading in data

Example:

```
# create dataframe and write out as a .csv file:
dataframe <- data.frame(cbind(rep(c("blue", "red", "green"),3), c(1:9)))
dataframe <- write.csv(dataframe, "dataframe.csv")

# count fields in .csv file:
count.fields(file="dataframe.csv", sep=",", skip=1)
```

```
## [1] 3 3 3 3 3 3 3 3 3
```

Explanation of Example: This says that there are 3 values on each of the 9 line of this .csv file we read in.

cummax

Morgan W. Southgate

The function cummax returns a vector whose elements are the cumulative maxima of the elements in the argument as read from L to R.

cummax(x)

Input:
- x: a numeric or complex object

Output:
- an atomic vector whose elements are the cumulative maxima of the elements in the argument as processed from L to R. Processing from L to R, the function consecutively finds the largest number. If the number in position x+1 is larger than the number in position x, its numeric value becomes the next element in the vector. If the number in position x+1 is smaller than the number in position x, the numeric value of x becomes the next element in the vector.

```
# Create a list d by concatenating sequences 1:3, 0:2, and 2:4
d <- c(1:3, 0:2, 2:4)

# View list d
print(d)
```

```
## [1] 1 2 3 0 1 2 2 3 4
```

```
# Find the cumulative maxima of the list
cummax(d)
```

```
## [1] 1 2 3 3 3 3 3 3 4
```

cummin()

Alexander Looi

This function apparently has nothing to do with the spice, which is tragic. Instead it goes element by element, starting with the first element, through a vector evaluating if successive elements are the smallest values up until that point in the vector. Additionally, it tracks the min values in a vector. For example consider the vector c(1, 4, 5, 2, 0, 1, -1), first element is 1 so 1 is stored as the first element. The second element is 4 and is larger than 1, so 1 (the first element) is retained as the smallest element and is also stored as the second element. This repeats until the end of the vector.

```
ran_num1 = c(23, 20, 16)
ran_num2 = c(4, 11, 6)
ran_num3 = c(7, 8, 22)
ran_num4 = c(10, 11, 12)
cummin(c(ran_num1, ran_num2, ran_num3, ran_num4))
```

```
##  [1] 23 20 16  4  4  4  4  4  4  4  4  4
```

```
# TRUE and FALSE can be considered 1 and 0 respectively.
ran_num4 = c(10, TRUE, FALSE)
cummin(c(ran_num1, ran_num2, ran_num3, ran_num4))
```

```
##  [1] 23 20 16  4  4  4  4  4  4  4  1  0
```

```
# chars can't be evaluated and are returned as NA's
ran_num4 = c('10', TRUE, FALSE)
cummin(c(ran_num1, ran_num2, ran_num3, ran_num4))
```

```
## Warning: NAs introduced by coercion
```

```
##  [1] 23 20 16  4  4  4  4  4  4  4 NA NA
```

cumprod

Samantha Alger

cumprod() returns a vector with elements that are the cumulative products

```
# create a vector consisting of 3 elements
x <-c(3,5,10)

# cumprod(x) will find the cumulative products
cumprod(x)
```

```
## [1]   3  15 150
```

```
# the cumprod() function is doing the following calculations:
3 * 1
```

```
## [1] 3
```

```
3 * 5
```

```
## [1] 15
```

```
3*5*10
```

```
## [1] 150
```

`cumsum`

Erin L. Keller

the cumsum function returns a vector which contains the cumulative sums of the elements of the input arguments (i.e. the first value in the output vector is the sum of the first element, the second value in the output vector is the sum of the first two elements, and so on). The input argument must be a numeric or complex object and the output is a numeric vector. If NA is included in the input argument, each sum from the position of NA on will be reported as "NA".

```
cumsum(1:10)
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```
a<-c(1,2,3,NA,4)
cumsum(a)
```

```
## [1]  1  3  6 NA NA
```

`data.frame`

Matthias Nevins

Description: The function data.frame() creates data frames, tightly coupled collections of variables which share many of the properties of matrices and of lists, used as the fundamental data structure by most of R's modeling software.

Arguments:

- row.names - "NULL" for a single integer or character sting specifying a column to be used as rownames. You can use a character or integer to give a specific rowname for the data frame.
- check.rows - if TRUE you can check to make sure the rowlength and names are consistent
- check.names - if TRUE you can check to make sure the rownames of the variables in the data frame are valid and to ensure there are no duplicates
- stringAsFactors - this logical argument asks if vectors should be converted to factors? The default is TRUE. Make stringAsFactors = FALSE to not convert vectors

```
# We explored a similar example in class. See below for an example of how to construct a data.fram
e

# Begin by creating variables
number <- 1:20
species <- rep(c("Maple", "Beech", "Pine", "Oak", "Spruce"), each=4)
basalArea <- runif(20)

dFrame <- data.frame(number,species,basalArea, stringsAsFactors = FALSE)
print(dFrame)
```

```
##    number species  basalArea
## 1       1   Maple 0.90695438
## 2       2   Maple 0.20981677
## 3       3   Maple 0.35813799
## 4       4   Maple 0.44829914
## 5       5   Beech 0.90642643
## 6       6   Beech 0.38943930
## 7       7   Beech 0.51745975
## 8       8   Beech 0.12523909
## 9       9    Pine 0.03014575
## 10     10    Pine 0.77180549
## 11     11    Pine 0.32741508
## 12     12    Pine 0.38947869
## 13     13     Oak 0.04105275
## 14     14     Oak 0.36139663
## 15     15     Oak 0.57097808
## 16     16     Oak 0.68488024
## 17     17  Spruce 0.97111675
## 18     18  Spruce 0.70195881
## 19     19  Spruce 0.01154550
## 20     20  Spruce 0.53553213
```

```
str(dFrame)
```

```
## 'data.frame':    20 obs. of  3 variables:
##  $ number   : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ species  : chr  "Maple" "Maple" "Maple" "Maple" ...
##  $ basalArea: num  0.907 0.21 0.358 0.448 0.906 ...
```

data.matrix()

Alex Burnham

Description: Converts a data frame into a numeric matrix. This means that character and factor variables will be coerced into numeric values (ranked) rather thatn following R's usual hierarchical structure.

Arguments:

- frame = a data frame whose components are logical vectors, factors or numeric vectors.
- rownames.force = TRUE or FALSE or NA. Should the matrix have character rownames rather than no names.

Example:

```
# make a dataframe:
dataframe <- data.frame(cbind(c("blue", "red", "green"), c(1:3)))
print(dataframe)
```

```
##      X1 X2
## 1  blue  1
## 2   red  2
## 3 green  3
```

```
# convert to a matrix
data.matrix(frame=dataframe)
```

```
##      X1 X2
## [1,]  1  1
## [2,]  3  2
## [3,]  2  3
```

Explanation of Example: dataframe was converted into a matrix (numeric) with the factor levels for color being used as their numeric values.

`diag()`

Alex Burnham

Description: Extracts or replaces the diagnals of a matrix. Value will can assign differnt values to the matrix. On its own, it will extract the diagnals which can be useful in multivariate analysis when dealing with correlation or variance/covariance matrices.

Arguments:

- x = a matrix
- value = either a single value or a vector of length equal to that of the current diagonal.
- nrow= optional number of rows when x isn't a matrix
- ncol = optional number of columns when x isn't a matrix

Example:

```
# make a 3 by 3 matrix of interger values 1 through 9:
matrix <- matrix(data=c(1:9), nrow=3)

# Returns the diagnal of this matrix
diag(x=matrix)
```

```
## [1] 1 5 9
```

```
# Replaces the diagnal with all 3s and prints matrix:
diag(x=matrix) <- 3
print(matrix)
```

```
##      [,1] [,2] [,3]
## [1,]    3    4    7
## [2,]    2    3    8
## [3,]    3    6    3
```

Explanation of Example: Output one takes the diagnal starting in the top left going down to the bottom right and returns the values. The second output replaces that same diagnal with a string of 3s.

`diff()`

Alex Burnham

Description: Returns suitably lagged and iterated differences between the values in a vector or a matrix.

Arguments:

- x = a numeric vector or matrix containing the values to be differenced
- differences = an integer indicating the order of the difference.
- lag = inger of which lag to use

Example:

```
# create vector of values 1 through 10:
x <- c(1:10)

# Differnces between values in that have a differnce of 2
diff(x=x, differences = 2)
```

```
## [1] 0 0 0 0 0 0 0 0
```

```
# Differnces between values in that have a differnce of 2
diff(x=x, differences = 1)
```

```
## [1] 1 1 1 1 1 1 1 1 1
```

Explanation of Example: The first output gives all 0 values because this sequece increases by 1 and no values next to eachother have differences of 2.The next output looks for differnces of 1. They are all differnt by 1 so it returns a string of 1s.

`dim`

Melanie R. Kazenel

The `dim` function can be used to obtain or specify the dimensions of an R object.

For use of `dim` to obtain the dimensions of an object, the function's input is an R object of more than one dimension, such as a matrix, array, or data frame. The output is a set of numbers indicating the dimensions of the object. For instance, when the input is a 2-dimensional object such as a matrix or data frame, the first number in the output is the number of rows in the object, and the second number is the number of columns.

For use of `dim` to specify the dimensions of an object, the input can be an R object of one or more dimensions. The `dim` function can be used to assign a new set of dimensions to the object, so long as the new set of dimensions is compatible with the number of observations in the object. The output in this case is an object with the newly specified dimensions.

```
# Example use of the `dim` function to obtain the dimensions of a matrix
m <- matrix(data = 1:8, nrow = 4, ncol = 2) # creates a matrix
print(m)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

```
dim(m) # prints out the dimensions of the matrix in the form of "rows, columns"
```

```
## [1] 4 2
```

```
# Example use of the `dim` function to convert a vector into a matrix of specified dimensions
m2 <- c(1:12) # creates a vector
print(m2)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
dim(m2) <- c(3,4) # converts the vector into a matrix with 3 rows and 4 columns
print(m2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

the '/' operator

Alexander Looi

This is used to divide two variables, vectors, numbers, datasets, matrices by each other. It won't return a faction if the result is not a whole number

```
# Single values
a = 1
b = 2
a/b
```

```
## [1] 0.5
```

```
# two vectors
a = 1:4
b = 5:8
a/b
```

```
## [1] 0.2000000 0.3333333 0.4285714 0.5000000
```

```
# vectors of different length. In this case a warning is thrown, and R recycles the first value of
 the shorter vector.
a = 1:4
b = 5:9
a/b
```

```
## Warning in a/b: longer object length is not a multiple of shorter object
## length
```

```
## [1] 0.2000000 0.3333333 0.4285714 0.5000000 0.1111111
```

```
# Char strings don't work!
# a = "a"
# b = 2
# a/b

# matrices and data.frames will work
a = matrix(1:4, 2, 2)
b = matrix(5:8, 2, 2)
a/b
```

```
##           [,1]      [,2]
## [1,] 0.2000000 0.4285714
## [2,] 0.3333333 0.5000000
```

```
# dataframes work as well
a = data.frame(matrix(1:4, 2, 2))
b = data.frame(matrix(5:8, 2, 2))
a/b
```

```
##          X1        X2
## 1 0.2000000 0.4285714
## 2 0.3333333 0.5000000
```

```
# Careful with dataframes, make sure there are not char strings in them!
a = data.frame(1:3, letters[1:3], stringsAsFactors = F)
b = data.frame(4:6, letters[4:6], stringsAsFactors = F)
# a/b # this will not work!

# careful you don't divide by 0 which will return a special value Inf or -Inf
# Also, sometimes R thinks of TRUE and FALSE as 1 or 0 so dividing by TRUE will be like dividing b
y 1, and dividing by FALSE will be like dividing by 0!
a = 45
b = TRUE
c = FALSE
a/b
```

```
## [1] 45
```

```
a/c
```

```
## [1] Inf
```

$

Peter Clark

The dollar sign (**$**) is an operator that can be used on matrices, arrays and lists and can "call for" or extract specific items by name in a data frame. This can be useful if you want to inspect a specific element of your dataframe (a column, a cell) or to search the output from a model.

Usage

x$name where *x* is the object from which to extract and *name* is a character string contained within *x*

```
# Call for a column in a dataframe

soil <- data.frame(1:14, 5) # create a dataframe
colnames(soil) <- c("pH","Ca") # assign column names
soil$pH # calls for the data in column titled "pH"
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

```
#----- Another example like the one above
x = list(a = rnorm(5), b = rnorm(7), c = rnorm(10))
str(x)
```

```
## List of 3
##  $ a: num [1:5] 0.98 -1.399 1.825 1.381 -0.839
##  $ b: num [1:7] -0.262 -0.0688 -0.3789 2.582 0.1298 ...
##  $ c: num [1:10] 0.2017 -0.0699 -0.0925 0.4489 -1.0644 ...
```

```
x$a
```

```
## [1]  0.9804641 -1.3988256  1.8248724  1.3812987 -0.8388519
```

```
#-------------------

# Calling for statistics (here, p-value in a linear model)
varY <- runif(10) # assign random numbers to x and y
varX <- runif(10)
myModel <- lm(varY~varX) # linear model running regresion using the 10 x,y, variables
names(summary(myModel)) # use this to search for other names in the lists in your model summary
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

```
summary(myModel)$coefficients # calls out the statistics from one list in the summary statistics.
```

```
##               Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)  0.8231706  0.1559533  5.278315 0.0007477293
## varX        -0.4999599  0.2739020 -1.825324 0.1053893346
```

```
# Since this is a matrix, you can call for an element (cell)
summary(myModel)$coefficients[2,4] # calls for the p value, found in the second row, forth column
```

```
## [1] 0.1053893
```

[[

Samantha Alger

double brackets will return only a single element from a list. A single bracket will return you a list with as many elements.

```
lst <- list('one','two','three')

a <- lst[1]
class(a)
```

```
## [1] "list"
```

```
## returns "list"


a <- lst[[1]]
class(a)
```

```
## [1] "character"
```

```
# returns "character"



#Example showing different uses with double and single brackets:
#Create a matrix
A = matrix(c(2,4,3,1,5,7),nrow=2,ncol=3,byrow=TRUE)
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    3
## [2,]    1    5    7
```

```
#Use double brackets to see the first element in the matrix
A[[1]]
```

```
## [1] 2
```

```
#Use single brackets to see the entire first column
A[,1]
```

```
## [1] 2 1
```

`dput`

Emily Mikucki

The `dput` function allows you to write a text representation of an R object (x) to a file or external connection. The format is put(x, file = "", control = "") where "file" is either a character string naming a file or an external connection ("" indicates the output in the console) and control is a character vector indicating the despersing options (there's a whole list of them under .deparseOpts) and include "keepNA", "keepInteger", and "showAttributes" which format the output accordingly. You can use more than one of these option by using the c() function.

The dput function goes hand-in-hand with dget (the output) where the format is dget(file, keep.source = FALSE). "keep.source" signifies if the source formatting should be retained from the dput.

R warns that the dput function is not a good way to transfer objects between R session. They say you should use the `dump` or `save` functions for that.

```
## Write an ASCII version of function mean to the file "foo"
dput(mean, "foo")
## And read it back into 'bar'
bar <- dget("foo")
## Create a function with comments
baz <- function(x) {
  # Subtract from one
  1-x
}
## and display it
dput(baz)
```

```
## function (x)
## {
##     1 - x
## }
```

```
## and now display the saved source
dput(baz, control = "useSource")
```

```
## function(x) {
##   # Subtract from one
##   1-x
## }
```

```
#This is the example R used under help, so it may not be the best....
#But "foo" should now be in your working directory!
```

=

Lauren Ash

The equal sign (=) can also be used as an assignment operator of a variable instead of <-. However, it is frowned upon in the coding community as the <- assignment operator is more universal and compatible with older versions of S-Plus. However, be careful because it usually does not work as an assignment operator within a function (see code below).

```
vecNum = c("1","2","3")
print(vecNum) ## The = works as an assignment
```

```
## [1] "1" "2" "3"
```

```
rm(x) # This removes x from the environment
median(x = 1:10)
```

```
## [1] 5.5
```

```
#print(x) # x assignment from within the function can NOT be used outside  function
#Error in print(x) : object 'x' not found
median(x <- 1:10)
```

```
## [1] 5.5
```

```
print(x) # X can be use outside the function with the <- operator
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

==

Peter Clark

The logical vector **==** is a relational operator that compares values for exact equality. It can be used on atomic vectors and results in a binary output (e.g., True False).

Useage

x == y compares two variabes for exact equality

```
x1 <- 5 - 3
x2 <- 4 - 1
x1 == x2    # is x1 exactly equal to x2? No, so R returns FALSE
```

```
## [1] FALSE
```

```
x1a <- 10-2
x2a <- 12-4
x1a == x2a # is x1A exactly equal to x2A? Yes, so R returns TRUE
```

```
## [1] TRUE
```

!=

April D. Makukhov

This specific operator translates in R as “not equal to” (the not operator being the ! component and = indicating “equal”). This is very useful for when you want an output from vectors and dataframes that are not equal to a numeric value.

```
# In this example, I am creating a vector 'x' that is made up of the following values
x <- c(0,1,2,3,0,4,5,0,8,2,0,0,1)

# Now I want to make a new vector 'y' that contains all values from except for 0, so I use the !=
 command to specifcy "not equal to zero".
y <- x[which(x != 0)]
y # notice that the output contains all the values of the x vector except for 0
```

```
## [1] 1 2 3 4 5 8 2 1
```

`!`

Emily Mikucki

The `!` in R is used as a logical negation or a “not operator”. It indicates the information you DO NOT want included in your clause, statement, function, etc. It can easily be used to subset certain data points or create clauses.

```
height.inches <- c(68, 65, 70, 71, 65)
height.inches >= 70 #This function shows the data points that are greater or equal to 70
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```
height.inches == 70 #This function shows the data points are exactly equal to 70
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

```
height.inches != 70 #This function shows all of the data points that are NOT equal to 70.
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
```

`exp`

Morgan W. Southgate

The exp function computes the exponential function for base e.

exp(x)

Input:
- x: a numeric or complex vector

Output:
- The result of e^x

```
# Find the result of raising e to the third power (e^3).
exp(3)
```

```
## [1] 20.08554
```

`expand.grid()`

Alexander Looi

This function creates a data frame from vectors or factors. These vectors or factors do not need to be the same length and can be of any type. What it will do is create a data frame with all combinations of the inputted vectors or factors. Columns are named as the orginal input vectors. To keep character strings in the dataset as characters set stringAsFactors = FALSE, otherwise the default is TRUE and character strings will be converted to type factor. A second argument ‘KEEP.OUT.ATTRS’ allows the user to decide if they want attributes to be calculated for use in “predict” methods (when using the produced table in the predict() function or other regressions).

```
a = seq(1,10, length = 20)
b = letters[1:5]
c = c('species_1', 'species_2')

expand.grid(a,b,c)
```

```
##         Var1 Var2      Var3
## 1    1.000000    a species_1
## 2    1.473684    a species_1
## 3    1.947368    a species_1
## 4    2.421053    a species_1
## 5    2.894737    a species_1
## 6    3.368421    a species_1
## 7    3.842105    a species_1
## 8    4.315789    a species_1
## 9    4.789474    a species_1
## 10   5.263158    a species_1
## 11   5.736842    a species_1
## 12   6.210526    a species_1
## 13   6.684211    a species_1
## 14   7.157895    a species_1
## 15   7.631579    a species_1
## 16   8.105263    a species_1
## 17   8.578947    a species_1
## 18   9.052632    a species_1
## 19   9.526316    a species_1
## 20  10.000000    a species_1
## 21   1.000000    b species_1
## 22   1.473684    b species_1
## 23   1.947368    b species_1
## 24   2.421053    b species_1
## 25   2.894737    b species_1
## 26   3.368421    b species_1
## 27   3.842105    b species_1
## 28   4.315789    b species_1
## 29   4.789474    b species_1
## 30   5.263158    b species_1
## 31   5.736842    b species_1
## 32   6.210526    b species_1
## 33   6.684211    b species_1
## 34   7.157895    b species_1
## 35   7.631579    b species_1
## 36   8.105263    b species_1
## 37   8.578947    b species_1
## 38   9.052632    b species_1
## 39   9.526316    b species_1
## 40  10.000000    b species_1
## 41   1.000000    c species_1
## 42   1.473684    c species_1
## 43   1.947368    c species_1
## 44   2.421053    c species_1
## 45   2.894737    c species_1
## 46   3.368421    c species_1
## 47   3.842105    c species_1
## 48   4.315789    c species_1
## 49   4.789474    c species_1
## 50   5.263158    c species_1
## 51   5.736842    c species_1
## 52   6.210526    c species_1
## 53   6.684211    c species_1
## 54   7.157895    c species_1
## 55   7.631579    c species_1
## 56   8.105263    c species_1
## 57   8.578947    c species_1
## 58   9.052632    c species_1
## 59   9.526316    c species_1
## 60  10.000000    c species_1
## 61   1.000000    d species_1
## 62   1.473684    d species_1
## 63   1.947368    d species_1
## 64   2.421053    d species_1
## 65   2.894737    d species_1
## 66   3.368421    d species_1
## 67   3.842105    d species_1
## 68   4.315789    d species_1
## 69   4.789474    d species_1
## 70   5.263158    d species_1
## 71   5.736842    d species_1
## 72   6.210526    d species_1
## 73   6.684211    d species_1
## 74   7.157895    d species_1
## 75   7.631579    d species_1
## 76   8.105263    d species_1
## 77   8.578947    d species_1
## 78   9.052632    d species_1
## 79   9.526316    d species_1
## 80  10.000000    d species_1
## 81   1.000000    e species_1
## 82   1.473684    e species_1
## 83   1.947368    e species_1
## 84   2.421053    e species_1
## 85   2.894737    e species_1
## 86   3.368421    e species_1
## 87   3.842105    e species_1
## 88   4.315789    e species_1
## 89   4.789474    e species_1
## 90   5.263158    e species_1
## 91   5.736842    e species_1
## 92   6.210526    e species_1
## 93   6.684211    e species_1
## 94   7.157895    e species_1
## 95   7.631579    e species_1
## 96   8.105263    e species_1
## 97   8.578947    e species_1
## 98   9.052632    e species_1
## 99   9.526316    e species_1
## 100 10.000000    e species_1
## 101  1.000000    a species_2
## 102  1.473684    a species_2
## 103  1.947368    a species_2
## 104  2.421053    a species_2
## 105  2.894737    a species_2
## 106  3.368421    a species_2
## 107  3.842105    a species_2
## 108  4.315789    a species_2
## 109  4.789474    a species_2
## 110  5.263158    a species_2
## 111  5.736842    a species_2
## 112  6.210526    a species_2
## 113  6.684211    a species_2
## 114  7.157895    a species_2
## 115  7.631579    a species_2
## 116  8.105263    a species_2
## 117  8.578947    a species_2
## 118  9.052632    a species_2
## 119  9.526316    a species_2
## 120 10.000000    a species_2
## 121  1.000000    b species_2
## 122  1.473684    b species_2
## 123  1.947368    b species_2
## 124  2.421053    b species_2
## 125  2.894737    b species_2
## 126  3.368421    b species_2
## 127  3.842105    b species_2
## 128  4.315789    b species_2
## 129  4.789474    b species_2
## 130  5.263158    b species_2
## 131  5.736842    b species_2
## 132  6.210526    b species_2
## 133  6.684211    b species_2
## 134  7.157895    b species_2
## 135  7.631579    b species_2
## 136  8.105263    b species_2
## 137  8.578947    b species_2
## 138  9.052632    b species_2
## 139  9.526316    b species_2
## 140 10.000000    b species_2
## 141  1.000000    c species_2
## 142  1.473684    c species_2
## 143  1.947368    c species_2
## 144  2.421053    c species_2
## 145  2.894737    c species_2
## 146  3.368421    c species_2
## 147  3.842105    c species_2
## 148  4.315789    c species_2
## 149  4.789474    c species_2
## 150  5.263158    c species_2
## 151  5.736842    c species_2
## 152  6.210526    c species_2
## 153  6.684211    c species_2
## 154  7.157895    c species_2
## 155  7.631579    c species_2
## 156  8.105263    c species_2
## 157  8.578947    c species_2
## 158  9.052632    c species_2
## 159  9.526316    c species_2
## 160 10.000000    c species_2
## 161  1.000000    d species_2
## 162  1.473684    d species_2
## 163  1.947368    d species_2
## 164  2.421053    d species_2
## 165  2.894737    d species_2
## 166  3.368421    d species_2
## 167  3.842105    d species_2
## 168  4.315789    d species_2
## 169  4.789474    d species_2
## 170  5.263158    d species_2
## 171  5.736842    d species_2
## 172  6.210526    d species_2
## 173  6.684211    d species_2
## 174  7.157895    d species_2
## 175  7.631579    d species_2
## 176  8.105263    d species_2
## 177  8.578947    d species_2
## 178  9.052632    d species_2
## 179  9.526316    d species_2
## 180 10.000000    d species_2
## 181  1.000000    e species_2
## 182  1.473684    e species_2
## 183  1.947368    e species_2
## 184  2.421053    e species_2
## 185  2.894737    e species_2
## 186  3.368421    e species_2
## 187  3.842105    e species_2
## 188  4.315789    e species_2
## 189  4.789474    e species_2
## 190  5.263158    e species_2
## 191  5.736842    e species_2
## 192  6.210526    e species_2
## 193  6.684211    e species_2
## 194  7.157895    e species_2
## 195  7.631579    e species_2
## 196  8.105263    e species_2
## 197  8.578947    e species_2
## 198  9.052632    e species_2
## 199  9.526316    e species_2
## 200 10.000000    e species_2
```

```
# Using the KEEP.OUT.ATTRS
# notice that the structures of these two tables are different. 'y' has attributes
# while 'z' has none. Notice the column names are not 'a' or 'b', but var1 var2.
a = 1:5
b = letters[1:3]
y = expand.grid(a,b)
z = expand.grid(a,b, KEEP.OUT.ATTRS = FALSE)

str(y)
```

```
## 'data.frame':	15 obs. of  2 variables:
##  $ Var1: int  1 2 3 4 5 1 2 3 4 5 ...
##  $ Var2: Factor w/ 3 levels "a","b","c": 1 1 1 1 1 2 2 2 2 2 ...
##  - attr(*, "out.attrs")=List of 2
##   ..$ dim     : int  5 3
##   ..$ dimnames:List of 2
##   .. ..$ Var1: chr  "Var1=1" "Var1=2" "Var1=3" "Var1=4" ...
##   .. ..$ Var2: chr  "Var2=a" "Var2=b" "Var2=c"
```

```
str(z)
```

```
## 'data.frame':	15 obs. of  2 variables:
##  $ Var1: int  1 2 3 4 5 1 2 3 4 5 ...
##  $ Var2: Factor w/ 3 levels "a","b","c": 1 1 1 1 1 2 2 2 2 2 ...
```

`factorial`

Emily Mikucki

The `factorial` operator computes the factorial of a number or multiplies a series of descending numbers. The factorial can be found for any numeric vector x.

```
factorial(1) # = 1 x 1
```

```
## [1] 1
```

```
factorial(3) # = 3 x 2 x 1
```

```
## [1] 6
```

```
factorial(8) # = 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1
```

```
## [1] 40320
```

```
factorial(c(4,3,2)) #you can also do it for lists of numbers where it will generate separate value
s for each variable
```

```
## [1] 24  6  2
```

`floor`

Samantha Alger

Floor(x) rounds *down* a given number (x) to an integer.

```
floor(2.76) # the function rounds down 2.76 to the value '2'
```

```
## [1] 2
```

```
floor(-3.456) # the function rounds down -3.456 to the value '-4'
```

```
## [1] -4
```

format

Emily Mikucki

The `format` function formats an R object “x” (typically numeric but can really be anything) into a nice printer-friendly version. There are a lot of default settings; I will only go through a few:

Default for spacing between parentheses is “trim = FALSE”; this means that all of the logical, numeric and complex values are right justified, so if the largest value or variable has 3 digits and the smallest only has 1, then the smaller one will have extra spaces between the " “.

The “justify” argument similarly sets the spacing of variables/names that are of unequal length. You can use “right”, “left”, “center” or “none”. The default is justify = left.

Using the “scientific” arguments sets whether something is written in scientific notation. The default is scientific = FALSE

The “nsmall” argument sets the number of digits allowed past the decimal point. The default is nsmall = 0 and I believe the maximum is nsmall = 20

```
#Trimming the spacing between " "
format(1:10)
```

```
##  [1] " 1" " 2" " 3" " 4" " 5" " 6" " 7" " 8" " 9" "10"
```

```
format(1:10, trim = TRUE)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

```
#Setting the justification
zz <- data.frame("(row names)"= c("aaaaa", "b"), check.names = FALSE)
format(zz)
```

```
##   (row names)
## 1       aaaaa
## 2           b
```

```
format(zz, justify = "left")
```

```
##   (row names)
## 1       aaaaa
## 2       b
```

```
#Using the scientific argument
format(2^31-1)
```

```
## [1] "2147483647"
```

```
format(2^31-1, scientific = TRUE)
```

```
## [1] "2.147484e+09"
```

```
#Using nsmall for decimals
format(13.7)
```

```
## [1] "13.7"
```

```
format(13.7, nsmall = 3)
```

```
## [1] "13.700"
```

```
format(c(6.0, 13.1), digits = 2)
```

```
## [1] " 6" "13"
```

```
format(c(6.0, 13.1), digits = 2, nsmall = 1)
```

```
## [1] " 6.0" "13.1"
```

get()

Alexander Looi

the get() function takes a character string as input and will retrieve the corresponding variable of that same name. Usually used in conjunction with the assign() function. This function is useful when looping through variables of different names, or if you need code to be flexible and self create variable names. It will allowyou to store these variables in a seperate vector and retrieve them later.

```
a = runif(100, 1, 10)
b = runif(100, 0, 10)

# takes in a character string
get('a')
```

```
##   [1] 8.138253 3.935373 9.621387 6.877176 5.128082 6.454197 3.598497
##   [8] 7.460478 9.310577 7.074455 2.673460 4.132604 2.111676 1.973046
##  [15] 3.681579 8.543799 9.934940 4.934210 2.823081 9.674844 6.943484
##  [22] 3.683051 2.073360 6.397764 2.081715 8.095888 4.326365 9.606906
##  [29] 9.222190 8.410027 3.875339 8.899303 8.203159 6.502210 1.651790
##  [36] 4.794289 4.100190 7.765621 2.968762 3.629206 4.203580 6.687657
##  [43] 9.032780 7.696295 5.115002 1.324859 6.116631 4.951143 6.404360
##  [50] 9.562445 3.424882 6.929546 1.688241 1.641567 4.332984 3.673920
##  [57] 5.959704 4.329015 8.611206 6.587116 4.592260 3.696954 4.430079
##  [64] 7.310214 9.521788 7.955455 2.975951 7.444993 6.977962 7.356252
##  [71] 3.516652 7.410254 6.948479 1.369114 1.550273 3.517855 3.717709
##  [78] 9.609497 4.508368 4.346099 8.588282 8.625115 3.873980 2.192554
##  [85] 6.553982 8.121741 4.042438 9.149028 2.778010 8.146767 7.791426
##  [92] 9.202554 3.904189 1.775525 9.200804 9.596564 7.040286 7.704840
##  [99] 4.960153 2.035355
```

```
# you can use a vector with character strings. This will allow you to loop through a vector of var
iables.
var_names = c('a', 'b')
get(var_names[1])
```

```
##   [1] 8.138253 3.935373 9.621387 6.877176 5.128082 6.454197 3.598497
##   [8] 7.460478 9.310577 7.074455 2.673460 4.132604 2.111676 1.973046
##  [15] 3.681579 8.543799 9.934940 4.934210 2.823081 9.674844 6.943484
##  [22] 3.683051 2.073360 6.397764 2.081715 8.095888 4.326365 9.606906
##  [29] 9.222190 8.410027 3.875339 8.899303 8.203159 6.502210 1.651790
##  [36] 4.794289 4.100190 7.765621 2.968762 3.629206 4.203580 6.687657
##  [43] 9.032780 7.696295 5.115002 1.324859 6.116631 4.951143 6.404360
##  [50] 9.562445 3.424882 6.929546 1.688241 1.641567 4.332984 3.673920
##  [57] 5.959704 4.329015 8.611206 6.587116 4.592260 3.696954 4.430079
##  [64] 7.310214 9.521788 7.955455 2.975951 7.444993 6.977962 7.356252
##  [71] 3.516652 7.410254 6.948479 1.369114 1.550273 3.517855 3.717709
##  [78] 9.609497 4.508368 4.346099 8.588282 8.625115 3.873980 2.192554
##  [85] 6.553982 8.121741 4.042438 9.149028 2.778010 8.146767 7.791426
##  [92] 9.202554 3.904189 1.775525 9.200804 9.596564 7.040286 7.704840
##  [99] 4.960153 2.035355
```

>

Peter Clark

The greater than operator **>** is a relational operator that allows for the comparison of atomic vectors. The output is binary (e.g., True False). This symbol may be combined with other operators (relational, assignment) to further refine your comparison.

Usage

x > y

```
# How it works
x <- 1:10 # produces a sequence of numbers
x > 8  # of variable x, lets see how many values are greater than 8. Produces a TRUE/FALSE output
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

```
x < 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
#---------
# may be used to compare two values, where
x <- 5
y <- 16
x<y
```

```
## [1] TRUE
```

```
x>y
```

```
## [1] FALSE
```

```
x<=5 # > may be combined with other symbols to refine your comparison. Here we examine "greater th
an or equal to"
```

```
## [1] TRUE
```

```
# see help function for complete list of relational operators
```

>=

Emily Mikucki

The `>=` operator is the greater than or equal to argument and uses the arguments x and y which can be atomic vectors, symbols, calls or other objects. For example, x >= y, signifies that x can be greater or equal to y. When posing this argument in R for idenitifying if certain variables fit the argument, you will receive TRUE or FALSE statements. This function can also be used for creating clauses, etc.

```
height.inches <- c(68, 65, 70, 71, 65)
height.inches >= 70 #This function shows the data points that are greater or equal to 70
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```
x <- matrix(1:9,3,3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
which(x >= 3)
```

```
## [1] 3 4 5 6 7 8 9
```

head

Morgan W. Southgate

The head function returns the first part of a vector, matrix, table, data frame, or function.

head(x, n= …)

Input:
- x: a vector, matrix, table, data frame, or function. - n: a single integer which (if positive) specifies the size of the resulting object - the number of elements for a vector, the number of rows for a matrix or data frame, or the number of lines for a function. If negative, n specifies all but the n last elements of x.

Output:
- The first section of x, the extent of which is specified by n.

```
# Create sequence data for infilling matrix
x <- seq(from=5, to=100, by=5)

# Create a matrix from sequence x
xmatrix <- matrix(data=x, nrow=10, ncol=2, byrow=TRUE)

# View the first 3 rows of matrix x
head(xmatrix, n=3)
```

```
##      [,1] [,2]
## [1,]    5   10
## [2,]   15   20
## [3,]   25   30
```

identical

Melanie R. Kazenel

The `identical` function can be used to test whether two objects are exactly equal to one another. The function takes two R objects of any type as input. The output is TRUE if the objects are equal to one another, and FALSE otherwise. The "num.eq" argument can be added to specify whether double and complex non-NA numbers should be compared using "==" (equal), which is the default (num.eq = TRUE), or bitwise (num.eq = FALSE). The "single.NA" argument can be used to specify whether to differentiate different types of NAs and NaNs; the default is TRUE, signifying that they should not be differentiated.

```
# Check whether two vectors are equal to one another
z <- c(2,3,5,7,"ten", "seven")
x <- c(2,3,5,7,"ten", "seven")
y <- c(2,3,5,7,"ten", "eight")
identical(z,x) # z and x are equal to one another
```

```
## [1] TRUE
```

```
identical(z,y) # z and y are not equal to one another
```

```
## [1] FALSE
```

`intersect()`

Alex Burnham

Description: This function is one of four Set Operators. It takes two arguments (x and y) each of them is an atomic vector. It checks the two vectors for numbers that are the same value and returns those numbers.

Arguments:

- x = your vector 1
- y = your vector 2

Example:

```
# create two vectors x and y
x <- c(3:13)
y <- c(8:18)

# use the intersect function:
intersect(x=x, y=y)
```

```
## [1]  8  9 10 11 12 13
```

Explanation of Example: In this example the vector x has values 3 through 13 and y has 8 through 18. Values 8 through 13 are returned because they overlap or intersect betwee the vectors x and y. ### `is.na` #### Matthias Nevins `is.na` is a generic function that indicates which elements are missing within a vector. "NA" is a logical constant of length 1 which contains a missing value indicator.

```
isX <- c(1,2,NA,4,5,NA,6)
is.na(isX) # is.na indicates which elemenst are missing
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

```
is.na(c(1,4,8,2,4,NA,6,NA,9))
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```
xx <- c(0:4)
is.na(xx) <- c(2, 4)
is.na(xx)
```

```
## [1] FALSE  TRUE FALSE  TRUE FALSE
```

is.character

Morgan W. Southgate

The is.character function tests if an object is of character type.

is.character(x)

Input:
- x: the object to be tested

Output:
- Test of character type for x in logical format. TRUE result means that the object of character type, FALSE result means that object not of character type.

```
# Create an atomic vector e consisting of the first four lowercase letters in the alphabet
e <- letters[1:4]

# Test if the vector e is of character type
is.character(e)
```

```
## [1] TRUE
```

```
# Create an atomic vector f consisting of a number sequence 1-10
f <- 1:10

# Test if the vector f is of character type
is.character(f)
```

```
## [1] FALSE
```

is.finite()

Alexander Looi

This function tests for Inf or -Inf and returns TRUE if the object being evaluated is neighter Inf or -Inf.

```
# returns True
a = 5
is.finite(10)
```

```
## [1] TRUE
```

```
is.finite(a)
```

```
## [1] TRUE
```

```
# returns False
is.finite(Inf)
```

```
## [1] FALSE
```

```
is.finite(-Inf)
```

```
## [1] FALSE
```

```
is.finite(NA)
```

```
## [1] FALSE
```

```
# can take vectors
nums = runif(20, 1, 20)
nums[which(nums < 3)] = Inf
nums
```

```
##  [1]  8.294514  4.909840  3.636827  8.392561  6.054878 14.362629  8.749685
##  [8]  6.045720  8.618579  4.748950 16.768702 11.031767  8.507474 11.905365
## [15] 19.448469 13.352847  7.338422  7.297501 18.906263  8.241137
```

```
# and will returns a vector of true and false. It returns true when an element is finite, and fals
e when the cell is not.
is.finite(nums)
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [15] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
# character values will also return false, so be careful since an error will not be thrown.
nums[which(nums > 18)] = 'a'
is.finite(nums)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

`is.logical()`

Alexander Looi

This is a function that tests if a single variable, atomic vector, matrix, or data.frame has logical variables (boolean) within it. This is regardless of the actual value of the boolean. So TRUE and FALSE variables when passed through is.logical will both return as TRUE. All other variable types will return as FALSE. Additionally vectors with multiple variables types, even if they include booleans will return FALSE since the *entire* variable is not a boolean.

```
# case where we are evaluating single variables
g = TRUE
f = FALSE
# both return TRUE
is.logical(g); is.logical(f);
```

```
## [1] TRUE
```

```
## [1] TRUE
```

```
# Passing other kinds of variables
n = 1
j = 0
h = 'k'
# These all return FALSE
is.logical(n); is.logical(j); is.logical(h)
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
# Passing a vector of TRUE FALSE
nums = runif(20)
TF_vec = c(nums > 0.5)

# Returns True
is.logical(TF_vec)
```

```
## [1] TRUE
```

```
# Passing a vector with different kinds of variables in it.
mixed_vec = c(TF_vec, "gg", 1)
is.logical(mixed_vec)
```

```
## [1] FALSE
```

`is.numeric`

Erin L. Keller

is.numeric determines whether a particular variable is numeric or not. The only input needed is the object to be tested (variable) and the output will be a logical vector with “TRUE” indicating that the variable is numeric (double or integers) and “FALSE” indicating that the variable is not numeric. It is important to note that while double is identical to numeric, however, is.double is not the same as is.numeric.

```
x<-4
y<-"Yes"
is.numeric(x)
```

```
## [1] TRUE
```

```
is.numeric(y)
```

```
## [1] FALSE
```

`length`

Peter Clark

The **length** function provides or allows you to set the length of an object (vector,lists) in the form of an integer. Note: If the parameter is a matrix or dataframe, it returns the number of variables (columns)

Usage

length(x) returns the length of a variable as an integer

length(x) <- value assigns the length of an object (vector, list), where “value” is a non-negative integer. Decimals (doubles) will be rounded down

```
# assign the length of a vector of 10, trucating at xth component
x<-1:100
length(x) <- 10
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# examine a vector is by returning its length (count)
x<-1:100
length(x)
```

```
## [1] 100
```

```
# derive the length of two vectors to look at differences
z<-(1:30)
p<-(1:20)
length(z)-length(p)
```

```
## [1] 10
```

<

Matthias Nevins

R funtion "<" is a *RELATIONAL OPERATION*. This binary operation can be used to compare atomic vectors.

```
# e.g. You could assign a variable "a" to a random set of numbers and then determing which of thos
e numbers are less than a selected value
runif(10)
```

```
##  [1] 0.15563612 0.96384166 0.00126048 0.70752621 0.63018760 0.77307804
##  [7] 0.89260688 0.51120408 0.74900107 0.92637980
```

```
a <- runif(10)
a < .05
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
# Or you could create two variables and compare them
x<-5
y<-6
x<y
```

```
## [1] TRUE
```

<=

Erin L. Keller

<= is an inequality sign that can be used to determine if the value to the left of the carat is smaller than or equal to the value to the right of the equal sign. The input arguments can be integers, atomic vectors, etc. and the ouput that is given is a logical vector where "TRUE" indicates that the inequality is correct and "FALSE" indicates that the inequality is incorrect.

```
a<-1
b<-2
a<=b
```

```
## [1] TRUE
```

```
b<=a
```

```
## [1] FALSE
```

`<<-`

Lauren Ashlock

This function assigns a value to a variable. The arguments used are the variable and a value to be assigned to x. The difference between this function and the <- function is that <<- will search through parent environments of a package you are working in for the variable you have called. If it finds the variable in the parent environment, it will assign the value for the variable in the parent environment to the variable in your global environment. If it does not find the variable, then assignment takes place in the global environment.

```
#example
outer_func <- function(){
   inner_func <- function(){
       a <<- 30
       print(a)
   }
   inner_func()
   print(a)
}

outer_func()
```

```
## [1] 30
## [1] 30
```

```
print(a)
```

```
## [1] 30
```

`list`

April D. Makukhov

This function is used to create a list in R, which can be made up of numeric values or even variable names.

```
y<-list(1,2,3,4)
y
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
```

```
x<-list('apple','orange','strawberry','kiwi')
x
```

```
## [[1]]
## [1] "apple"
##
## [[2]]
## [1] "orange"
##
## [[3]]
## [1] "strawberry"
##
## [[4]]
## [1] "kiwi"
```

```
z<- c(x,y) # here is an example where I am combining the 2 lists I made above into one vector

z
```

```
## [[1]]
## [1] "apple"
##
## [[2]]
## [1] "orange"
##
## [[3]]
## [1] "strawberry"
##
## [[4]]
## [1] "kiwi"
##
## [[5]]
## [1] 1
##
## [[6]]
## [1] 2
##
## [[7]]
## [1] 3
##
## [[8]]
## [1] 4
```

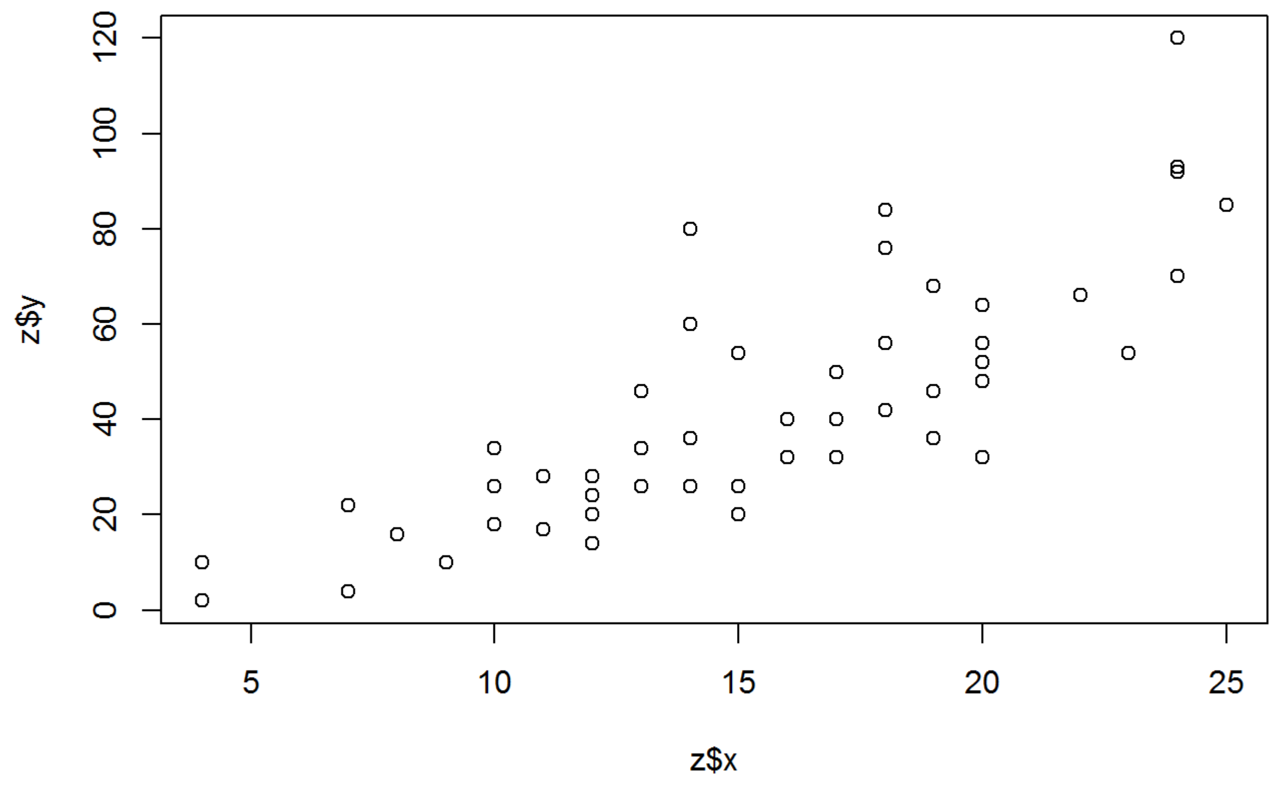
```
# list is also a useful function for plotting. In this example below, we are using data that is al
ready in R (car data), and creating a vector z that is making a list made up of components 'x' and
 'y', and each of those components is equal to a column of data from the cars dataset.

z <-list(x=cars[,1], y=cars[,2]) # making a list with 2 components, x and y, where x is comprised
 of the cars data in column 1 of the dataset, and y is comprised of the cars data in column 2 of t
he dataset

z
```

```
## $x
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
## [24] 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24
## [47] 24 24 24 25
##
## $y
##  [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34
## [18]  34  46  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76
## [35]  84  36  46  68  32  48  52  56  64  66  54  70  92  93 120  85
```

```
plot(z) # after creating this vector z, we can see the 2 components of the list, z$x and z$y, plot
ted against each other
```



`load`

Lauren Ashlock

- The load funtion loads R objects that have already been saved in R. The arguments for this function are the file, envir, and verbose.
- The input for the file argument is a connection or character string that gives the name of the file you want to reload.
- The envir argument designates the environment where you want the data to be loaded. This can be useful for avoiding any overwriting of existing objects with the same name in the current environment. It is best to use envir= to load the object into a different environment.
- The verbose argument designates whether or not you want item names to be printed as they load.This argument can be useful for debugging. The output of this function is a character vector of the names of objects created.
- This is particularly useful for saving and reloading .Rdata or .Rhistory files (saving your workspace) into your current environment.

```
#example
save.image(file= "homework_6.RData") #save your workspace

#To load this make sure you are in the current working directory
#in which you saved this workspace then run the following code

load(file= "homework_6.RData")

#the default settings are to load the file in the global environment and to have verbose = False.
 In order to debug...

load(file= "homework_6.Rdata",verbose= TRUE)
```

```
## Loading objects:
##   model
##   varX
##   varY
##   var_names
##   mixed_vec
##   number
##   dog
##   xmatrix
##   cat
##   height.inches
##   soil
##   species
##   TF_vec
##   DF
##   bar
##   outer_func
##   c1
##   mat
##   ran_num1
##   x1
##   ran_num2
##   x2
##   ran_num3
##   baz
##   ran_num4
##   a
##   xx
##   b
##   matrix2
##   c
##   d
##   e
##   f
##   A_c
##   g
##   h
##   nums
##   j
##   k
##   basalArea
##   m
##   combine
##   n
##   x1a
##   p
##   test
##   q
##   dFrame
##   t
##   .Random.seed
##   y_1
##   vecNum
##   y_2
##   m2
##   myModel
##   x
##   matrix
##   y
##   dataframe
##   z
##   lst
##   num
##   x2a
##   q_c
##   zz
##   isX
##   A
```

`log()`

Alex Burnham

Description: The log() function takes two arguemnts: a single value or a vector (x) and "base", which allows you to choose the base of your log. By default, log() transforms the vector using the base e or natural logarithm which is defined in R as exp(1).

Arguments:

- x = a single value or vector
- base = the base of the log you want to use (default = exp(1) or natural log)

Example:

```
# create vector of values 1 through 10:
x <- c(1:10)

# take the log (base e) of this vector x:
log(x=x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

```
# take the log (base 10) of this vector x:
log(x=x, base=10)
```

```
##  [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513 0.8450980
##  [8] 0.9030900 0.9542425 1.0000000
```

Explanation of Example: The first output is the log (base e) transformation of the vector x (1 through 10) The second output is the log (base 10) transformation of the vector x (1 through 10) ### `log10` #### April D. Makukhov

This function computes the logarithm of a numeric vector, specifically computing in base 10. As we've learned previously in mathematics with logarithms, the input and output are switched; that is, when computing log10(x), we are actually specifying the output value 'x' and want to know what input value 'y' is the exponent with base 10 that gives us value x. Thus, we provide the 'output' in a sense by giving the expected value 'x' and are in return given the 'input', which is the exponent and the answer to our logarithmic computation.

```
# We can find the value of log10(x) as follows:
log10(10)
```

```
## [1] 1
```

```
# Here, we see the output from above is 1. This is because base 10 (which is the base we are in be
cause we are saying 'log10' instead of just 'log') to an exponent=1 is equal to 10, the value in o
ur ()

# Another example
log10(100) # 10^2 =100, so the output of this should be 2, as the output from a logarithic computa
tion IS the exponent.
```

```
## [1] 2
```

```
# If we want to take the logarithm of a different base, we would just change the 10 value and coul
d do so as follows:
log(10, base = 3) # Here we are specificying we are in base 3, instead of 10
```

```
## [1] 2.095903
```

```
# base 10 and base 2 are generally the more common logarithmic bases, which is why log10 and log 2
 are both already functions. But, we could use the above example with base 10 and still get the sa
me vale as log10. For example:
log(10, base = 10)
```

```
## [1] 1
```

```
log(100, base =10)
```

```
## [1] 2
```

`log2()`

Alex Burnham

Description: The log2() function takes a single argument (x) which is a single value or a vector and log transforms it using the base 2 logarithm.

Arguments:

- x = a single value or vector

Example:

```
# create vector of values 1 through 10:
x <- c(1:10)

# take the log (base 2) of this vector x:
log2(x=x)
```

```
##  [1] 0.000000 1.000000 1.584963 2.000000 2.321928 2.584963 2.807355
##  [8] 3.000000 3.169925 3.321928
```

Explanation of Example: The output is the log (base 2) transformation of the vector x (1 through 10)

match

Matthias Nevins

Description: The 'match' function returns or finds the first occurance of the first argument in the second argument. The %in% function can also be used in a similar way to return a logical vector (TRUE,FALSE)

- Usage: match(x,table,nomatch=NA_integer_,incomparables = NULL) or x %in% table
- Arguments:
    - x: a vector or NULL and represent the values to be matched
    - table: a vector or NULL and represent the values to be matched against the first argument
    - nomatch: the value to be returned if there is no match *incomparables: defines the a vector of the values that cannot be matched. FALSE=NULL

EXAMPLE: Using 'match' and '%in%'

- Argument 1: x=3
- Argument 2: table = 2:6

```
match(x=3, table=2:6)
```

```
## [1] 2
```

```
# The match function returns the first occurance of the first argument (x=3), in the second argume
nt (table = 2:6). In this example the vector '3' is found in the second position of the table
# If you would like to return a logical vector you can use %in%
3%in%2:6 # This asks if 3 is in your table 2:6. It returns "TRUE" indictating that the first argum
ent is indeed found in the second
```

```
## [1] TRUE
```

matrix

Morgan W. Southgate

The matrix function creates a matrix from the given set of values.

matrix (data = NA, nrow=1, ncol=1, byrow=FALSE, dimnames = NULL)

Input:
- data: the data vector to be converted to matrix form
- nrow: the desired # of rows
- ncol: the desired # of columns
- byrow: a logical input. FALSE (default) fills matrix by columns, TRUE fills by rows.
- dimnames: NULL or a list of of length 2 giving the row and column names, respectively. An empty list is treated as NULL and a list of length one as the row names only. NULL assigns row and column numbers automatically using matrix format of (row#,column#).

Output:
- A matrix of the given data.

```
# Make an atomic vector of 16 values using a sequence function
a <- 1:16

# Assign dimension names by creating a list - first four lowercase letters of alphabet will become
 row names, first four uppercase letters will become column names
dimnames1 <- list(letters[1:4],LETTERS[1:4])

# Assemble the atomic vector data into a matrix with 4 rows, 4 columns, and fill matrix by row
m1 <- matrix(data = a, nrow=4, ncol=4, byrow=TRUE, dimnames = dimnames1)
print(m1)
```

```
##    A  B  C  D
## a  1  2  3  4
## b  5  6  7  8
## c  9 10 11 12
## d 13 14 15 16
```

max()

Alexander Looi

This function returns the maximum value in a vector, matrix or data frame. You can use the na.rm variable to tell the function to remove NA's and NaN's. If there are either of these null types in your data set and you do not specify their removal with na.rm and will return NA.

```
# returns the largest number
nums = runif(100, 1, 100)
max(nums)
```

```
## [1] 99.68448
```

```
# does this even if there are negatives
nums = runif(100, -100, 100)
max(nums)
```

```
## [1] 96.6014
```

```
# watch out for NA's and Inf will return NA
nums[which(nums < -75)] = NA
max(nums)
```

```
## [1] NA
```

```
max(nums, na.rm = T)
```

```
## [1] 96.6014
```

```
# it interprets -Inf properly
nums[which(is.na(nums))] = -Inf
max(nums)
```

```
## [1] 96.6014
```

```
max(nums, na.rm = T)
```

```
## [1] 96.6014
```

`mean`

Melanie R. Kazenel

The `mean` function calculates the arithmetic mean of a set of values. The function takes a numeric or logical vector (or a date, date-time, or time interval object) as input, and the output of the function is the arithmetic mean of the values in the object. The output is in the form of a vector with a length of one. In addition, the `mean` function can take a complex vector as its input if specific parameters are specified.

Under default settings, the `mean` function calculates the mean of the all of the observations in an object, and any NA values in the object are not removed prior to the calcuation of the mean. To remove a particular fraction of the observations from each end of the object before the mean is computed, the “trim” argument can be added. To remove NAs from the object before the mean is computed, the “na.rm” argument can be added.

```
### Example use of the `mean` function for a numeric vector

data <- c(4,8,10,25)

# Calculate the mean using default settings
mean(data)
```

```
## [1] 11.75
```

```
# Trim 25% of the observations from each end of the vector, and then calculate the mean of the rem
aining observations.
mean(data, trim = 0.25)
```

```
## [1] 9
```

```
### Example use of the `mean` function for a vector containing a "NA" value

data2 <- c(4,8,10,25,NA)

# Calculating the mean using default settings yields "NA"
mean(data2)
```

```
## [1] NA
```

```
# Adding "na.rm = TRUE" removes the "NA" and then calculates the mean of the remaining observation
s.
mean(data2, na.rm = TRUE)
```

```
## [1] 11.75
```

`median`

Erin L. Keller

The purpose of this function is to **calculate the median** value of a vector. The median value of a set of number is the value at the midpoint of the vector, so there are equal amounts of items on either side of the midpoint. In the case of a vector with an even number of values, the median function will take the average of the two midpoint values. The input for this function is a numerical vector and the ouput is an integer.

To use this function, you will need two arguments:
* x - the numeric vector from which you want the median
* na.rm - logial value specifying whether NA values should be included or discarded in the calculation
* na.rm = FALSE will not include NA values (default)
* na.rm = TRUE will include NA values

```
z <- 1:10
print(z)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
median(z, na.rm=FALSE)
```

```
## [1] 5.5
```

`message`

Matthias Nevins

The message function generates a diagnostic message from the argument you define. These messages are not warnings or errors, but nevertheless represented as condition.

- Usage: message(…, domain = NULL, appendLF = TRUE)
- Arguments:
    - …: zero or more objects which displayed as the message. Multiple objects will be pasted together with no seperator.
    - domain: If NA, messages will not be translated
    - appendLF: Logical argument should messages

```
message("ABC","DEFGH") # Both arguments are not separated
```

```
## ABCDEFGH
```

```
message("good morning everyone", appendLF = FALSE) # you can display a separated message by includ
ing the whole message within a single object
```

```
## good morning everyone
```

`min`

Emily Mikucki

The `min` operator returns the minimum value of all of the values in a given numeric data set/argument. The default setting includes na.rm = FALSE, so NAs will be included in the output. For example, a non numeric input will result in a NA output.

```
min(5:1, pi) #Gives the minimum number of all of the numbers in this set
```

```
## [1] 1
```

```
height.inches <- c(68, 65, 70, 71, 65)#Similary gives the minimum of this data set
min(height.inches)
```

```
## [1] 65
```

`-`

Erin L. Keller

The - symbol, or the “minus sign” subtracts the value after the “-” from the value before the “-” as it does in arithmetic and reports the difference between those two values. The input value can be a vector, data frame, or matrix and the ouput value will be integers in a vector, matrix, or data frame (matching the input format).

```
a<-matrix(1:6,3,2)
b<-matrix(3:8,3,2)
c<-b-a
print(c)
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    2    2
## [3,]    2    2
```

%%

Morgan W. Southgate

The %% function is an arithmetic operator function which gives the modulus of two arguments. This modulus operator divides the x argument by the y argument, rounds down to the nearest integer, and then finds the remainder between the (integer value * y) and the value of x. For vectors x & y where x is an integer multiple of y, the result of x mod y is 0.

x%%y

Input:
- x,y: numeric or complex vectors or objects which can be coerced to this type

Output:
- the difference between x and the rounded integer quotient of x/y

```
# Assign integer values to the vectors x & y
x <- 15
y <- 4

# Find the difference between the value of x and the value of the rounded integer quotient x/y. 1
5/4 = 3.75, rounded down to 3. Then 3*4 = 12, and 15-12 =3.
x%%y
```

```
## [1] 3
```

```
# When y is an integer multiple of x, x%%y = 0
12%%4
```

```
## [1] 0
```

*

Morgan W. Southgate

The function * is an arithmetic operator function which returns the product of two numeric or complex variables or vectors. If multiplying two vectors with multiple values in each together, the multiplication function finds the consecutive products of the values in the vectors from left to right.

x*y

Input:
- x,y : numeric or complex vectors or objects which can be coerced to numeric or complex vectors

Output:
- the product of x and y. Logical variables are coereced to integer or numeric vectors, where FALSE = 0 and TRUE=1.

```
# Create two vectors x1 & y1
x1 <- c(1,2,3)
y1 <- c(4,5,6)

# Find the products of the values contained in the vectors x1 and y1
x1*y1
```

```
## [1]  4 10 18
```

names

Melanie R. Kazenel

The `names` function can be used to obtain the names associated with the elements of an object. It can also be used to assign names to the object. The function's input is an R object, and the names associated with each element are the output.

```
# The following vector does not have names assigned to it
z <- c(1:4)
print(z)
```

```
## [1] 1 2 3 4
```

```
names(z)
```

```
## NULL
```

```
# Assign names to the vector using the "names" function
names(z) <- c("kale", "broccoli", "cabbage", "brussels sprouts") # assign names to the vector
print(z)
```

```
##             kale         broccoli          cabbage brussels sprouts
##                1                2                3                4
```

```
names(z) # obtain the names associated with the vector
```

```
## [1] "kale"             "broccoli"         "cabbage"
## [4] "brussels sprouts"
```

```
# Change the name assigned to a specific element within a vector by specifying the position of the
 element you wish to change the name of
z <- "names<-"(z, "[<-"(names(z), 2, "collards"))
print(z)
```

```
##             kale         collards          cabbage brussels sprouts
##                1                2                3                4
```

```
names(z)
```

```
## [1] "kale"             "collards"         "cabbage"
## [4] "brussels sprouts"
```

```
# Remove all names associated with the vector
names(z) <- NULL
print(z)
```

```
## [1] 1 2 3 4
```

ncol

Morgan W. Southgate

The ncol function returns the number of columns present in an atomic vector, matrix, data frame, or data array. Atomic vectors are treated as one-column matrices.

ncol(x)

Input:
- x: a vector, matrix, array, or data frame

Output:
- the number of columns in x

```
# Create sequence data for infilling matrix
b <- 1:12

# Create matrix with default number of columns
m2 <- matrix(data=b)

# Tell number of columns in m
ncol(m2)
```

```
## [1] 1
```

nrow

Erin L. Keller

nrow will report the number of rows of an array. The input arguments can be a vector, array, or data frame and the output will be a single integer reporting the number of rows.

```
matrix<-matrix(1:10,2,5)
print(matrix)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```
nrow(matrix)
```

```
## [1] 2
```

"|" symbol or operator

Alexander Looi

The pipe or bar in R denotes the “or” operator. This symbol is largely used in logical arguments where multiple booleans evaulations are given. Seperating the boolean evaluations with “|” means that only one of the evaluations have to return True for the entire evaluation to return True. All evaluations have to return False for the entire evaluation to return False. The best way to think about this is to ilterally think of it as the word “or”. So, if x is True or y is True or z is True, return True

```
x = 8
y = 10

# this will return true since one of the evaluations will return true.
y == 10 | x > 100
```

```
## [1] TRUE
```

```
# All evaluations have to return true have to return False for the entire evaluation to return Fal
se.
y == 11 | x >= 10
```

```
## [1] FALSE
```

%in%

Alexander Looi

This is an operator that compares two vectors and returns to the user, a boolean vector of where in the first vector, elements are matched in the second vector. This useful when users have two vectors with similar data and they want to see which elements in the longer vector match elements in the shorter vector. This operator is particularly useful with the which() function.

```
a = rep(letters[1:5], 20)
b = rep(letters[1:2], 4)

TF = a %in% b

TF_v = which(TF)

a[TF_v]
```

```
##  [1] "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a"
## [18] "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b" "a" "b"
## [35] "a" "b" "a" "b" "a" "b"
```

%/%

Emily Mikucki

The `%/%` operator is used for integer division (binary) and basically rounds down to the next integer below. It can also be explained as the remainder of x divided by y (x mod y). The arguments include x and y which are both numeric or complex vectors.

```
x <- -1:12
x %/% 5
```

```
##  [1] -1  0  0  0  0  0  1  1  1  1  1  2  2  2
```

+

Erin L. Keller

The + in R is used to indicate the addition of numbers as it is used in arithmetic. Using + is simple and only requires two values to be added together. Spaces between the values and the + are optional. The input value can be a vector, data frame, or matrix and the ouput value will be integers.

```
a<-4.4
b<-1.2
c<-a+b
print(c)
```

```
## [1] 5.6
```

```
d<-1:2
e<-d+1
print(e)
```

```
## [1] 2 3
```

pmax

April D. Makukhov

The ‘p’ in pmax stands for ‘parallel’. This function outputs the parallel maxima of values inputed. Essentially, this means you can take one or more vectors and, with pmax, create a single vector that has only the maxima between the multiple vectors. pmax allows you to find the maximum values among multiple vectors and receive an output with those maximum values.

```
# Making 2 vectors, x and y, of the same length
x <- c(1,5,7,9,3,15,37,128,4)
y <- c(2,8,12,54,207,95,73,122,62)
pmax(x,y) # this gives a single output vector that contains the maximum values of these two vector
s with respect to position. For example, the first value of the pmax vector is 2 because '2' in th
e first position of vector y is greater than '1' in the first position of vector x.
```

```
## [1]   2   8  12  54 207  95  73 128  62
```

pmin()

Alexander Looi

Not to be confused with the min() function. pmin() returns the min values of *parallel* input values. Parallel in this case means corresponding element positions of multiple vectors. For example, consider vectors a = c(“a”, “b”, “c”, “d”) and b = c(1, 2, 3, 4). The first elements of vectors a and b (“a” and 1) considered to be parallel elements. You can use the na.rm variable to tell the function to remove NA’s and NaN’s. If there are either of these null types in your data set and you do not specify their removal with na.rm the NA will be chosen as the “min” value.

```
ran_num1 = c(1, 20, 16)
ran_num2 = c(4, 11, 6)
ran_num3 = c(7, 8, 22)
ran_num4 = c(10, 11, 12)
# should return c(1, 8, 6)
pmin(ran_num1, ran_num2, ran_num3, ran_num4)
```

```
## [1] 1 8 6
```

```
# An NA in the data set
ran_num1 = c(1, 20, 16)
ran_num2 = c(4, 11, 6)
ran_num3 = c(7, NA, 22)
ran_num4 = c(10, 11, 12)
pmin(ran_num1, ran_num2, ran_num3, ran_num4)
```

```
## [1]  1 NA  6
```

```
# An NA in the data set but na.rm is used
ran_num1 = c(1, 20, 16)
ran_num2 = c(4, 11, 6)
ran_num3 = c(7, NA, 22)
ran_num4 = c(10, 11, 12)
pmin(ran_num1, ran_num2, ran_num3, ran_num4, na.rm = T)
```

```
## [1]  1 11  6
```

```
# careful with evaluating matrices! It will look at parallel elements between matrices.
ran_num1 = c(1, 20, 16)
ran_num2 = c(4, 11, 6)
ran_num3 = c(7, 8, 22)
ran_num4 = c(10, 11, 12)
a = matrix(c(ran_num1, ran_num2, ran_num4), 3, 3)
b = matrix(c(ran_num2, ran_num4, ran_num1), 3, 3)
pmin(a, b)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    1
## [2,]   11   11   11
## [3,]    6    6   12
```

```
# works for dataframes as well (should have the same result as the above matrix.
a = data.frame(ran_num1, ran_num2, ran_num4)
b = data.frame(ran_num2, ran_num4, ran_num1)
pmin(a, b)
```

```
##   ran_num1 ran_num2 ran_num4
## 1        1        4        1
## 2       11       11       11
## 3        6        6       12
```

```
# careful you don't use characters
a = data.frame(ran_num1, c('a', 'b', 'c'), ran_num4)
b = data.frame(ran_num2, ran_num4, ran_num1)
pmin(a, b)
```

```
## Warning in Ops.factor(left, right): '>' not meaningful for factors
```

```
##   ran_num1 c..a....b....c.. ran_num4
## 1        1                a        1
## 2       11                b       11
## 3        6                c       12
```

```
# even if the columns are aligned to the correct variable type
a = data.frame(ran_num1, c('a', 'b', 'c'), ran_num4, stringsAsFactors = F)
b = data.frame(ran_num2, c('g', 'g', 'h'), ran_num1, stringsAsFactors = F)
pmin(a, b)
```

```
##   ran_num1 c..a....b....c.. ran_num4
## 1        1                a        1
## 2       11                b       11
## 3        6                c       12
```

`print`

Erin L. Keller

print is a function that displays the ouput of the input object while making the arguments invisible. Typically, this function is used to visualize data and the same result can be achieved by simply typing the variable name. For this function, inputs of any form can be used; however, some forms may require additional information including: * quote - a logical object used to indicate whether strings should be printed * max.levels - an integer indicating how many levels for a factor should be printed. The detault is NULL which will print on one line of a specified width * digits - the minimum number of significant digits * na.print - a character string which indicates NA values in the output * zero.print - this character can specify if 0's should be included (some people prefer looking at decimals without the 0) * justify - a character that indicates whether strings should be right- or left-justified or not justified * useSource - this logical indicates whether an internally stored source should be present when printing (keep.source=TRUE if in use)

print.factor allows further customization whiel print.table allows further customization for tables. The output of this function will be the form of the input object.

```
Rainfall <- matrix(c(1.2,2.4,4.2,6.3,8.9,11.1,12.2,10.7),4,2)
colnames(Rainfall)<-c("Desert","Deciduous")
rownames(Rainfall)<-c("December","February","March","April")
print(Rainfall)
```

```
##          Desert Deciduous
## December    1.2       8.9
## February    2.4      11.1
## March       4.2      12.2
## April       6.3      10.7
```

`prod`

Emily Mikucki

The `prod` function is a multiplier operator. It returns the multiplication results of all the values present in the arguments you create (… represents numeric, complex or logical vectors). The default setting includes na.rm = FALSE where NAs are still included in your output.This is an easy way to multiply more than two variables at a time, rather than just using `*`.

```
x <- c(3.2,5,4.3) #You can create a list of values and multiply them this way.
prod(x)
```

```
## [1] 68.8
```

```
prod(4:6) #Or you can create a series and multiply them directly like this
```

```
## [1] 120
```

?

Morgan W. Southgate

The documentation shortcut ? provides access to documentation and help on an unknown function. Available for use only in the console.

?function.name

Input:
- name of the function

Output:
- documentation and user help displayed in the help console (lower R)

```
?mean
```

```
## starting httpd help server ...
```

```
##  done
```

range()

Alex Burnham

Description: range returns a vector containing the minimum and maximum of all the given arguments.

Arguments:

- first value is any numeric or character object
- na.rm = TRUE or FALSE (if TRUE NAs are omitted)

Example:

```
# create vector of values 1 through 10:
x <- c(1:10)

# find the range in vector x
range(x)
```

```
## [1]  1 10
```

Explanation of Example: The smallest value “1” and largest “10” are given.

`rbind`

Lauren Ashlock

- rbind takes a sequence of vector, matrix, or data-frame areguments and combines them by rows.
- Arguments: are vectors or matrices, that can be given as named arguments.
- deparse.level is an integer argument that controls the construction of labels in the case of non-matrix-like arguments. The default is 0, which constructs no labels. 1 or 2 constructs labels from the argument names
- make.row.names is an argument that is only used when binding data frames. It is a logical argument that indicates if unique and valid row.names should be constructed fromt the arguments.
- stringsAsFactors is a logical argument that is passed to as.data.frame. This only has an effect when the arguments contain a non-data.frame) character

```
#example

m<- rbind(1,1:7)
m
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    1    1    1    1    1    1
## [2,]    1    2    3    4    5    6    7
```

```
m <- rbind(m, 8:14)
m
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    1    1    1    1    1    1
## [2,]    1    2    3    4    5    6    7
## [3,]    8    9   10   11   12   13   14
```

```
#deparse.level argument

dd<- 10
rbind(1:4, c=2, "a++"=10, deparse.level =0) #this names the rows of the middle two arguments
```

```
##     [,1] [,2] [,3] [,4]
##        1    2    3    4
## c      2    2    2    2
## a++   10   10   10   10
```

```
rbind(1:4, c=2, "a++"=10, dd, deparse.level=1) #this names the last three arguments (this is the d
efault setting)
```

```
##     [,1] [,2] [,3] [,4]
##        1    2    3    4
## c      2    2    2    2
## a++   10   10   10   10
## dd    10   10   10   10
```

```
rbind(1:4, c=2, "a++" =10, dd, deparse.level=2) #takes the names from each element
```

```
##     [,1] [,2] [,3] [,4]
## 1:4    1    2    3    4
## c      2    2    2    2
## a++   10   10   10   10
## dd    10   10   10   10
```

`readLines`

Melanie R. Kazenel

The `readLines` function can be used to read text or data into R that is not formatted in a way conducive to being read in using a function such as `read.csv` or `read.table`. For instance, `readLines` can be used to read in unformatted text. The input for the function is a URL or file. The output is a vector in which each element corresponds to a line in the input file.

The “n=” argument can be used to specify the number of lines you want to be read in; the default value is -1 and means that all lines will be read in. The “ok” argument can be used to specify whether a warning message should come up if the end of the file is reached before the number of lines specified in the “n=” argument is reached; the default is TRUE. The “encoding” argument can be used to specify the type of encoding used in the document; the default is “unknown.” The “skipNul” argument can be used to specify whether nulls in the dataset should be skipped rather than read in; the default is FALSE. The “warn” argument can be added to specify whether certain warning messages should come up; the default is TRUE.

```
# Read all of the text from a webpage into R
z <- readLines("https://gotellilab.github.io/Bio381/CourseMaterials/CourseSyllabus.html")
summary(z)
```

```
##    Length     Class      Mode
##       284 character character
```

```
# Read the first 5 lines from a webpage into R
z <- readLines("https://gotellilab.github.io/Bio381/CourseMaterials/CourseSyllabus.html", n = 5L)
summary(z)
```

```
##    Length     Class      Mode
##         5 character character
```

```
print(z)
```

```
## [1] "<!DOCTYPE html>"
## [2] ""
## [3] "<html xmlns=\"http://www.w3.org/1999/xhtml\">"
## [4] ""
## [5] "<head>"
```

`readRDS`

Lauren Ashlock

This function is used to write a single R object to a file, and to restore that object. Arguments: - object: R object to be written to a file
- file: a connection or the name of the file where the R object is saved to or read from
- ascii: A logical. If TRUE or NA, an ASCII representation is written. The default for this argument is FALSE. This results in a binary representation.
- version: This argument will only be relevant with versions newer than the default version (2).
- compress: A logical stating whether or not you want your saved object to be compressed. The default for this argument is TRUE.
- refhook: This is a hook function for handling reference objects. The default for this argument is NULL.

```
#example

##save a single object to file
copepod<-10
saveRDS(object=copepod, file="copepod.rds")
##restore it under a different name
copepod2 <- readRDS(file="copepod.rds")
identical(copepod, copepod2)
```

```
## [1] TRUE
```

`read.csv`

Lauren Ashlock

This function reads in a file that is in table format, and creates a data frame from it. Default settings: read.csv(file, header = TRUE, sep = ",", quote = """, dec = ".", fill = TRUE, comment.char = "", …)

Arguments: - file: this is the file you want to be read from
- header: This is a logical value stating whether or not the file you are reading in contains the header names in the first line
- sep: This describes howyour characters are separated the default is space, but you can also use ","
- quote:This argument denotes howyour character strings are delimited. The default is to delimit strings by double quotes but you can also delimit strings by single quotes. If you want to disable quotes you can use quote = ""
- dec: The character used in the file for decimal points
- fill: a logical. If TRUE, then if your rows have unequeal length, blank fields are added.
- comment.char: A character vector of length one containing a single character or empty string. The default is to use "" . This turns off the interpretation of comments.

```
#example
#read.csv(file="OASV2_BodyLengthMeasurements_Oct2016.csv", header = TRUE, sep =",", quote="\"", de
c=".", fill=TRUE, comment.char="")
```

`read.delim`

Samantha Alger

read.delim() is used to read in delimited text files, where data is organized in a data matrix with rows representing cases and columns representing variables. read.delim(file,header=TRUE,sep="")

- file -A file location
- header - Whether the first line describes the column names
- sep - The table delimiter, often a tab () or comma

```
#The following is an example for how you would read in a .txt file.
#It is commented out since the .txt file does not exist

#d <-read.delim("annual.txt", header= TRUE, sep="\t")
```

`read.fwf`

Erin L. Keller

The purpose of read.fwf to to read a table of **f**ixed **w**idth **f**ormatted data into a data.frame. To use this function, you will need multiple arguments including: * the name of the file containing the data * the width of the fixed-dwitch fields in the form of a vector * a header containing a logical value specifying names of the variables * if this is present, the names of the variables must be delimited by sep * a character not used in the data set that will be the separator (sep) * row and column names (row.names and col.names) * n is the maximu number of lines to be included * skip can be used to identify how many rows should be skipped when reading the fwf file. * buffersize indicates the maximum number of lines to be read at one time (reducing this may reduce memory use when dealing with large files leading to faster processing)

The output of read.fwf will be a data frame as produced by read.table.

```
x <- read.fwf(
  file=url("http://www.cpc.ncep.noaa.gov/data/indices/wksst8110.for"),
  skip=4,
  widths=c(12, 7, 4, 9, 4, 9, 4, 9, 4))

head(x)
```

```
##             V1   V2   V3   V4   V5   V6   V7   V8  V9
## 1  03JAN1990   23.4 -0.4 25.1 -0.3 26.6  0.0 28.6 0.3
## 2  10JAN1990   23.4 -0.8 25.2 -0.3 26.6  0.1 28.6 0.3
## 3  17JAN1990   24.2 -0.3 25.3 -0.3 26.5 -0.1 28.6 0.3
## 4  24JAN1990   24.4 -0.5 25.5 -0.4 26.5 -0.1 28.4 0.2
## 5  31JAN1990   25.1 -0.2 25.8 -0.2 26.7  0.1 28.4 0.2
## 6  07FEB1990   25.8  0.2 26.1 -0.1 26.8  0.1 28.4 0.3
```

`rep`

Lauren Ash

The function rep() replicates the values in x (a vector of any mode including a list). Some common arguments are 'times', 'length.out', and 'each' which allows you to replicate your vector in a variety of ways.

```
rep(1:4, times=2)   # the argument times=2 consecutively lists 1-4 twice
```

```
## [1] 1 2 3 4 1 2 3 4
```

```
rep(1:4, each = 2)  # not the same! each=2 lists 2 1's, 2 2's, 2 3's etc.
```

```
## [1] 1 1 2 2 3 3 4 4
```

```
rep(1:4, times= c(2,1,2,1)) # you can vary how many times you want to repeat
```

```
## [1] 1 1 2 3 3 4
```

```
rep(rep(1:4, times=c(2,1,2,1)), each= 2) # you can use nested reps too
```

```
##  [1] 1 1 1 1 2 2 3 3 3 3 4 4
```

```
rep(1:4, each = 2, length.out = 4) # you can only show first 4 integers with the length.out argume
nt
```

```
## [1] 1 1 2 2
```

`rep_len`

Samantha Alger

rep_len() replicates values for a desired length ("len")

```
# the following code will repeat the three bird types 20 times
rep_len(c("finch","thrush","warbler"),20)
```

```
##  [1] "finch"   "thrush"  "warbler" "finch"   "thrush"  "warbler" "finch"
##  [8] "thrush"  "warbler" "finch"   "thrush"  "warbler" "finch"   "thrush"
## [15] "warbler" "finch"   "thrush"  "warbler" "finch"   "thrush"
```

`rev`

Samantha Alger

rev() reverses an R object, including vector, array, etc.

```
# First create a vector
x <- c("red","orange","blue")

#check it out
print(x)
```

```
## [1] "red"    "orange" "blue"
```

```
#now reverse the order of the vector and print
rev(x)
```

```
## [1] "blue"   "orange" "red"
```

`rle`

Erin L. Keller

rle standard for "Run Length Encoding" and determines the lengths and values of runs of equal values in a vector. The input argument must be an atomic vector and the ouput will indicate the lengths of the runs (integer) and the values of the run (numeric). The function, inverse.rle can also be used to transform the output of rle back into a vector.

```
a<-c(1,1,2,2,2,2,3,3,3,4,5,5,5,6)
rle(a)
```

```
## Run Length Encoding
##   lengths: int [1:6] 2 4 3 1 3 1
##   values : num [1:6] 1 2 3 4 5 6
```

```
b<-inverse.rle((rle(a)))
print(b)
```

```
##  [1] 1 1 2 2 2 2 3 3 3 4 5 5 5 6
```

`round()`

Alexander Looi

This function is used to “round” the under of significant digits in a numeric variable. It generally takes two arguments, the variable to be rounded and the number of digits to “round off”. If no digits are given the function removes all decimals.

```
a = runif(10, 0, 20)
round(a)
```

```
##  [1] 12  1  9 16 12  6 20 10 14 15
```

```
# round off 3 digits
round(a, digits = 3)
```

```
##  [1] 12.004  0.799  9.475 15.539 11.597  6.440 19.868  9.925 13.629 14.758
```

```
# It cannot take strings even if they are numbers
# round(as.character(a)) # can't be done

# also will not work
words = c('cat', 'dog', 'hat', 23.555)
# round(words)
```

`rownames`

Melanie R. Kazenel

The `rownames` function can be used to specify or obtain the names associated with rows in a matrix or matrix-like object. The function’s input is a matrix-like R object, and the names associated with each row are the output. The arguments “do.NULL” and “prefix” can be added; see the example below for an explanation of how to use these arguments.

```
# The following matrix does not have row names assigned to it
z <- matrix(data = c(1:8), nrow = 4, ncol = 2)
print(z)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

```
rownames(z) # under default settings, the output of 'rownames' is NULL when no row names have been
 assigned
```

```
## NULL
```

```
rownames(z, do.NULL = FALSE) # when do.NULL = FALSE, names are assigned to each row using the pref
ix "row" plus a number
```

```
## [1] "row1" "row2" "row3" "row4"
```

```
rownames(z, do.NULL = FALSE, prefix = "specialname") # the prefix argument can be added to specify
 that a name other than "row" should be added before each number when do.NULL = FALSE
```

```
## [1] "specialname1" "specialname2" "specialname3" "specialname4"
```

```
# Assign row names to the matrix
rownames(z) <- c("kale", "broccoli", "cabbage", "brussels sprouts") # assign row names to the matr
ix
print(z)
```

```
##                  [,1] [,2]
## kale                1    5
## broccoli            2    6
## cabbage             3    7
## brussels sprouts    4    8
```

```
rownames(z) # obtain the row names associated with the matrix
```

```
## [1] "kale"             "broccoli"         "cabbage"
## [4] "brussels sprouts"
```

`sample`

Samantha Alger

sample() will take a random sample of a specified size. You can specify whether to sample with or without replacement.

sample(x, size, replace =FALSE)

- x - object to be sampled
- size - size of the sample
- replace - should sampling be with replacement?

```
#create an object to sample
x <- runif(1:10)

# take a sample of 100 from x
# without replacement (we will never find duplicates)
sample(x, 4, replace = FALSE)
```

```
## [1] 0.0790970 0.7545179 0.3663761 0.7481242
```

```
# if you sample with replacement, you can sample at a size larger than the original object
#in this case, there will be numbers repeated
sample(x, 11, replace = TRUE)
```

```
##  [1] 0.0790970 0.9169217 0.4453595 0.4414792 0.7545179 0.4453595 0.7481242
##  [8] 0.4414792 0.9076130 0.0790970 0.9076130
```

save

April D. Makukhov

This functions allows the user to save objects in R into an external Rdata file that can then be used later in R via functions such as ‘load’ or ‘attach’. save.image() is an extension or shortcut of the ‘save’ function that allows the use to save the current workspace in R to be able to load it back in later. Basically, if you want to save a set of specific R objects in your code or to save your workspace, this is one way to do so.

```
x <- runif(10)
z <- list(a=1, b=2, c=TRUE, d=FALSE, e = "turtle")
save(x, z, file = "Example.RData")

# now, if you clear your R environment with the little broom icon, and use the following code belo
w to load in your Rdata file, you can go back to using the variables you used before, x and z, wit
hout having to re-run script. Essentially, the save function is saving your R environment.

load("Example.RData")
x
```

```
##  [1] 0.130109230 0.960689858 0.259571558 0.009066601 0.992870629
##  [6] 0.652903802 0.724484820 0.714281988 0.825024178 0.148244921
```

```
z
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] TRUE
##
## $d
## [1] FALSE
##
## $e
## [1] "turtle"
```

saveRDS()

Alexander Looi

This function allows the user to save text representations of objects they have created in R. These objects range from data frames, vectors, to linear models. These objects are saved directly to the current directory and do not need an extension to be saved (or later read by readRDS). The beauty of this function is that these saved objects can then be read in later and should still retain all of their original features. The two variables necessary to use this function are an R object, and a name for the file in the form of a character string. Additionally, you can specify if the user wants the object to be written in ascii format (a text format that is readable by most text editors). The user can specify version of R for the file to saved under (Do not save in versions < 2.0). The user can also compress the file to 'gzip', 'bzip2', or 'xz'.

```
y = runif(100)
x = 1:100

my_lm = lm(x ~ y)

saveRDS(my_lm, file = 'My_saveRDS')
list.files(pattern = 'My_saveRDS') # it should appear in the directory
```

```
## [1] "My_saveRDS"
```

```
my_lm_returned = readRDS('My_saveRDS')
my_lm
```

```
##
## Call:
## lm(formula = x ~ y)
##
## Coefficients:
## (Intercept)            y
##       57.79       -14.16
```

```
my_lm_returned
```

```
##
## Call:
## lm(formula = x ~ y)
##
## Coefficients:
## (Intercept)            y
##       57.79       -14.16
```

```
# can save data frames
MY_DF = data.frame(y, x)
saveRDS(MY_DF, file = 'My_DF')
readRDS('My_DF')
```

```
##              y  x
## 1  0.675003473  1
## 2  0.846872858  2
## 3  0.992584993  3
## 4  0.304093186  4
## 5  0.145394889  5
## 6  0.519036944  6
## 7  0.832152318  7
## 8  0.918171508  8
## 9  0.348103863  9
## 10 0.332689337 10
## 11 0.838867004 11
## 12 0.446239590 12
## 13 0.694593564 13
## 14 0.902810826 14
## 15 0.029474204 15
## 16 0.471807718 16
## 17 0.408890214 17
## 18 0.336940273 18
## 19 0.663275549 19
## 20 0.873135848 20
## 21 0.086759413 21
## 22 0.334643598 22
## 23 0.933748877 23
## 24 0.918592991 24
## 25 0.686753216 25
## 26 0.383543578 26
## 27 0.008982167 27
## 28 0.719938769 28
## 29 0.480926614 29
## 30 0.003722787 30
## 31 0.572111683 31
## 32 0.080655038 32
## 33 0.379890728 33
## 34 0.999523677 34
## 35 0.097467466 35
## 36 0.804216082 36
## 37 0.084229584 37
## 38 0.877182388 38
## 39 0.982542838 39
## 40 0.609853369 40
## 41 0.684833081 41
## 42 0.494595101 42
## 43 0.275989021 43
## 44 0.003721118 44
## 45 0.664426274 45
## 46 0.548279016 46
## 47 0.596033634 47
## 48 0.308346305 48
## 49 0.897289331 49
## 50 0.916157075 50
## 51 0.971690540 51
## 52 0.342376201 52
## 53  0.928428851  53
## 54  0.918632505  54
## 55  0.691446334  55
## 56  0.769923869  56
## 57  0.602196601  57
## 58  0.769321489  58
## 59  0.097983083  59
## 60  0.231752394  60
## 61  0.706835322  61
## 62  0.028821961  62
## 63  0.631142367  63
## 64  0.672157781  64
## 65  0.452885734  65
## 66  0.675898803  66
## 67  0.009393311  67
## 68  0.063480382  68
## 69  0.279233183  69
## 70  0.467886943  70
## 71  0.776457262  71
## 72  0.366924266  72
## 73  0.900720072  73
## 74  0.521405264  74
## 75  0.289609326  75
## 76  0.244583334  76
## 77  0.589133301  77
## 78  0.380756364  78
## 79  0.705413333  79
## 80  0.116040773  80
## 81  0.598454862  81
## 82  0.846073140  82
## 83  0.350864020  83
## 84  0.316167029  84
## 85  0.780196068  85
## 86  0.254789922  86
## 87  0.840200149  87
## 88  0.751392944  88
## 89  0.308782266  89
## 90  0.458046929  90
## 91  0.414898014  91
## 92  0.325016026  92
## 93  0.135616546  93
## 94  0.910110965  94
## 95  0.150384865  95
## 96  0.837922945  96
## 97  0.544273399  97
## 98  0.115952749  98
## 99  0.044229344  99
## 100 0.282693473 100
```

sd

Emily Mikucki

The `sd` function computes the standard deviation of the numeric vector x. You can use this for a vector you create in the console (i.e. using c()) or you can get the standard deviation of a variable you have defined in a csv file. The default settings are sd(x, na.rm = FALSE) where na.rm removes the NAs.

```
x <- c(179,160,136,227) #define your vector
sd(x) #equals 38.57892 using the R function
```

```
## [1] 38.57892
```

```
#If you're curious, this is the formula that is being used
n <- length(x)
sd1 <-sqrt(sum((x - mean(x))^2) / (n - 1))
sd1 #sd is still 38.57892
```

```
## [1] 38.57892
```

Seq

Erin L. Keller

Seq is the function for sequence generation and is a standard generic function. The arguments needed for this function are: * from, to - indicate what the starting and ending values of the sequence should be * by - the increment of the sequence (default = 1) * length.out - optional, the desired length of the sequence * along.with - takes the length from the length * : - can be used to create sequences of sequential numbers

The ouput for this function is a integer or double vector containing the desired sequence.

```
Sequence <-seq(c(1:10,20:30,40:50)) # multiple sequences can be concatenated together to form one
 sequence
print(Sequence)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32
```

seq_along

Matthias Nevins

seq_along generates regular sequences. seq_along(x) takes a vector for x, and it creates a sequence upto the count of elements in the vector. seq() acts like seq_along() except when passed a vector of length 1, in which case it acts like seq_len().

```
# EXAMPLE
a <- c(8, 9, 10)
b <- c(9, 10)
c <- 10

seq_along(a)
```

```
## [1] 1 2 3
```

```
seq_along(b)
```

```
## [1] 1 2
```

```
seq_along(c)
```

```
## [1] 1
```

```
# Compared to the seq() function

seq(a)
```

```
## [1] 1 2 3
```

```
seq(b)
```

```
## [1] 1 2
```

```
seq(c)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

`seq_len`

Erin L. Keller

Seq_len is a seqeunce generator that outputs a sequence of values from 1 to the value specified in the parentheses and is equivalent to seq(length.out=). Sequence length will automatically be inferred if the start and end values are given in the seq function (i.e. seq(from=1, to=20), sequence length will be 20). It is important to note that all numerical inputs should be finite.

```
a<-seq_len(10)
b<-seq_len(0)
print(a)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
print(b)
```

```
## integer(0)
```

`setdiff`

Morgan W. Southgate

The setdiff function is a set function which compares two sets of values and returns the values which are in the first set but NOT in the second set. Therefore, setdiff(x, y) returns a different output than setdiff(y, x).

setdiff(x, y)

Input:
- x, y: vectors (of the same mode) containing a sequence of items with no duplicate values.

Output:
- The values in the first set of values which are NOT in the second set of values.

```
# Create two sets of values set1 and set2 with overlapping but distinct values
set1 <- c( "a","b","d","e")
set2 <- c( "a","c","d","f")

# Use the setdiff function to return the values which are unique to set1
setdiff(set1,set2)
```

```
## [1] "b" "e"
```

```
# See that the output is changed by reversing the order of sets 1 and 2
setdiff(set2,set1)
```

```
## [1] "c" "f"
```

`setequal`

Samantha Alger

setequal() tests if two vectors are equal. The function will return a logical response: (TRUE or FALSE).

```
#create two different vectors using the sample function that will randomly sample 10 numbers from
 1 to 100.
x <- sample(1:100, size=10)
y <- sample(1:100, size=10)

#use setequal to check if the two vectors are the same. since we are using the sample function, we
 are randomly sampling and so the two vectors will be different. setequal returns a FALSE value.
setequal(x,y)
```

```
## [1] FALSE
```

```
#another example:
birds <- rep_len(c("finch","thrush","warbler"),20)
birds2 <- rep_len(c("NA","thrush","warbler"),20)

#using setequal returns a FALSE value because there is one value different
setequal(birds,birds2)
```

```
## [1] FALSE
```

sign

Lauren Ashlock

This function returns a vector with signs of the corresponding elements of a variable. The output gives you 1 for positive values, 0, or -1 for negative values. The only argument for this function is a numeric vector

```
MyVec <- c(0, 2, 5, 7, -1, -4, -7, 0, 5, -4)
sign(MyVec)
```

```
##  [1]  0  1  1  1 -1 -1 -1  0  1 -1
```

signif

Samantha Alger

signif(x, digits=y) rounds a number (x) to a specified number of significant digits (y)

```
signif(7.462527,digits=4)
```

```
## [1] 7.463
```

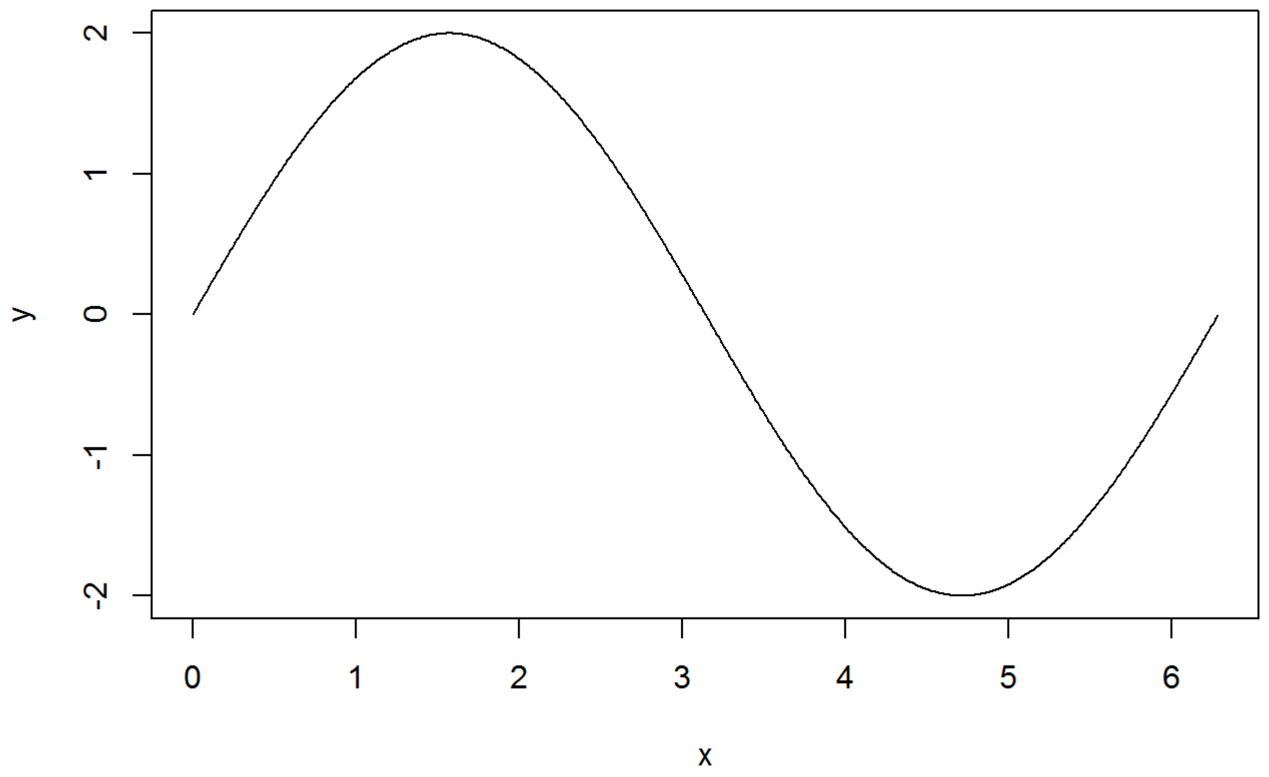
sin()

Alexander Looi

This is the trig-function sine. It takes in any numeric type variable and outputs a numeric type. The default "x" or input of this function are radians. To input degrees the user must convert radians to degrees.

```
x = seq(0, 2*pi, 0.01)
y = 2*sin(x)

# let's do the wave!
plot(x, y, type = 'l')
```



### sink ####Emily Mikucki

The `sink` function directs R output to an external source like a text file. To use `sink`, you should make a new.txt file and upload it to your working directory/R Project. This acts as your “file” argument. You might run into the arguments `append` and `split`. The default settings include append=FALSE and split=FALSE. The `append` option controls whether output overwrites or adds to a file. The `split` option determines if output is also sent to the screen as well as the output file. Note, `sink` will NOT redirect graphic ouputs (.jpg, .png, etc.).

```
sink("TEST.txt") #Here you upload that external file you have synched to your working directory/pr
oject
i <- 1:10 #Here you create functions, data, etc. that you want to send to that external file.
outer(i, i, "*") #This was just used to format the data in the file
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    4    6    8   10   12   14   16   18    20
##  [3,]    3    6    9   12   15   18   21   24   27    30
##  [4,]    4    8   12   16   20   24   28   32   36    40
##  [5,]    5   10   15   20   25   30   35   40   45    50
##  [6,]    6   12   18   24   30   36   42   48   54    60
##  [7,]    7   14   21   28   35   42   49   56   63    70
##  [8,]    8   16   24   32   40   48   56   64   72    80
##  [9,]    9   18   27   36   45   54   63   72   81    90
## [10,]   10   20   30   40   50   60   70   80   90   100
```

```
sink() #This function must be used to finish the sinking/synching process!
#Now if you open up TEST.txt outside of R, it will have all of the information that you directed t
o it.

sink("TEST.txt", append=FALSE, split=FALSE)
sink()
#Now all of that information is gone from the .txt file.
```

split

Emily Mikucki

The `split` function divides the data in a vector x into the groups defined by f. Basically it divides data into groups and then reassembles it The default setting is split(x, f, drop = FALSE, …). “x” can be a vector or data frame. “f” is a factor (in terms of “as.factor”) and defines the grouping from the split. “drop” is logical and indicates if levels that do not occur should be dropped (if f is a factor or a list).

```
#A simple example
A <- c(1,2,3,4)
B <- c("X","Y","X","Y")
sp <- split(A,B)
sp$X #see data split between Xs
```

```
## [1] 1 3
```

```
sp$Y
```

```
## [1] 2 4
```

```
## Split a matrix into a list by columns
ma <- cbind(x = 1:10, y = (-4:5)^2)
split(ma, col(ma))
```

```
## $`1`
##  [1]  1  2  3  4  5  6  7  8  9 10
## 
## $`2`
##  [1] 16  9  4  1  0  1  4  9 16 25
```

```
split(1:10, 1:2)
```

```
## $`1`
## [1] 1 3 5 7 9
## 
## $`2`
## [1]  2  4  6  8 10
```

sprintf

Peter Clark

This function allows for the creation of a character string that can be incoporated into a vector. It returns a character vector containing a formatted combination of text and variable values. While the paste function is more useful for vectors, **sprintf** is useful for precise control of the output. A wrapper for *c()* function

Usage

sprintf(fmt, …) where fmt is a character string up to 8192 characters and … are the values to be incorporated in the the fmt string

Subuse

- To substitute in a string or string variable, use *%s*
- For integers, use *%d* or a variant
- Many other options exist: For floating-point numbers, use *%f* for standard notation, and *%e* or *%E* for exponential notation. You can also use *%g* or *%G* for a “smart” formatter that automatically switches between the two formats, depending on where the significant digits are.

```
x <- 2349
y <- 1111
z <- 2
sprintf("Substitute in a string or number: %s", x)   # %s substitutes x into the string
```

```
## [1] "Substitute in a string or number: 2349"
```

```
sprintf("Can have multiple %s %E occurrences %s", x, z, y, "- got it?")
```

```
## [1] "Can have multiple 2349 2.000000E+00 occurrences 1111"
```

sqrt

Morgan W. Southgate

The sqrt function computes the square root of a value x or vector of values x.

sqrt(x)

Input:
- x: a numeric or complex vector or array

Output:
- the square root of x or the square root of each individual term in x

```
# Create a numeric vector v1
v1 <- c(16,4)

# Calculate the square roots of each integer value in v1
sqrt(v1)
```

```
## [1] 4 2
```

[

Samantha Alger

square brackets ('[') are used to reference or index a particular object within a vector, dataframe, or matrix.

```
# for vectors:
# create a vector
data = c(1,3,5,7,3,2)

#Using brackets will return the third value of the vector
data[3]
```

```
## [1] 5
```

```
#for a given dataframes or matrices:
# make a dataframe:
A <- 1:16
B <- rep(c("red","blue","green","NA"), each =4)
C <-runif(16)
data <- data.frame(A,B,C,stringsAsFactors = FALSE)

#check out the dataframe
print(data)
```

```
##     A     B          C
## 1   1   red 0.82496680
## 2   2   red 0.86319212
## 3   3   red 0.65435866
## 4   4   red 0.02200860
## 5   5  blue 0.99794305
## 6   6  blue 0.14764741
## 7   7  blue 0.08255771
## 8   8  blue 0.79769322
## 9   9 green 0.34406541
## 10 10 green 0.72860601
## 11 11 green 0.34104958
## 12 12 green 0.86083031
## 13 13    NA 0.40722344
## 14 14    NA 0.56856047
## 15 15    NA 0.90155034
## 16 16    NA 0.93526459
```

```
# Using brackets, the first value specifies the row, the second value specifies the column
#The following will return the value in the first row, second column of the dataframe:
data[1,2]
```

```
## [1] "red"
```

str

Peter Clark

str is a diagnostic function that allows you to compactly display the “structure” of an R object. This is a concise alternative to the summary function. Ideally, only one line for each ‘basic’ structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of lists. The basic goal of using str allows you to answer “whats in this object”?

Usage

str(object, …)

See ?help for greater detail of arguments

```
# examples include:

str(1:12) # show the structure of a sequence
```

```
##  int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
```

```
str(str) # show the arguement structure of a function
```

```
## function (object, ...)
```

```
# show the structure of a linear model
varY <- runif(10)
varX <- runif(10)
myModel <- lm(varY~varX)
str(myModel)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 0.325 0.408
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "varX"
##  $ residuals    : Named num [1:10] 0.3194 -0.0907 -0.2052 0.0603 0.3488 ...
##   ..- attr(*, "names")= chr [1:10] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:10] -1.6673 0.2177 -0.2995 0.0746 0.2528 ...
##   ..- attr(*, "names")= chr [1:10] "(Intercept)" "varX" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:10] 0.528 0.505 0.553 0.397 0.555 ...
##   ..- attr(*, "names")= chr [1:10] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:10, 1:2] -3.162 0.316 0.316 0.316 0.316 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:10] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "varX"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.32 1.1
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 8
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = varY ~ varX)
##  $ terms        :Classes 'terms', 'formula'  language varY ~ varX
##   .. ..- attr(*, "variables")= language list(varY, varX)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "varY" "varX"
##   .. .. .. ..$ : chr "varX"
##   .. ..- attr(*, "term.labels")= chr "varX"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(varY, varX)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "varY" "varX"
##  $ model        :'data.frame':   10 obs. of  2 variables:
##   ..$ varY: num [1:10] 0.847 0.414 0.347 0.457 0.904 ...
##   ..$ varX: num [1:10] 0.497 0.441 0.558 0.176 0.564 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language varY ~ varX
##   .. .. ..- attr(*, "variables")= language list(varY, varX)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "varY" "varX"
##   .. .. .. .. ..$ : chr "varX"
##   .. .. ..- attr(*, "term.labels")= chr "varX"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(varY, varX)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "varY" "varX"
##  - attr(*, "class")= chr "lm"
```

subset

Erin L. Keller

Subset is a function that pulls out data from the input object that meets the criteria of interest specified in the function. The input object can be vectors, data frames, and matrices and the arguments to be included are: * x - the object to be subsetted * subset - this is a logical expression where one specifies what elements to keep and to subset. Missing values are not included * select - an expression which indicates what columns to select from a data frame (only used for data frames and matrices) * drop - eliminates dimensions of an array that have only one level

The output to this function will be similar to the input object containing only the selected elements (vector) or rows/columns (matrix, data frame). It is important to note that some factors will have empty levels after subsetting and those that are unused will be removed.

```
a <- matrix(c(runif(10, min = 80, max = 100),(runif(10, min=85, max = 115))),nrow=10,ncol=2)
colnames(a)<-c("MaxTemp2015","MaxTemp2017")
print(a)
```

```
##       MaxTemp2015 MaxTemp2017
##  [1,]    99.74943   111.23041
##  [2,]    94.32902   107.30015
##  [3,]    88.22074    87.17898
##  [4,]    91.40487    97.77895
##  [5,]    82.09899   100.47322
##  [6,]    97.34537   107.16114
##  [7,]    89.55756    93.04061
##  [8,]    85.57133    99.61898
##  [9,]    89.26041   113.87186
## [10,]    80.82170   110.25655
```

```
subset(a, a[,1]>90 & a[,2]>100) # this will subset the original data matrix so that MaxTemp2015 wi
ll only diplay values over 90 while MaxTemp2017 will only display values greater than 100.
```

```
##      MaxTemp2015 MaxTemp2017
## [1,]    99.74943    111.2304
## [2,]    94.32902    107.3001
## [3,]    97.34537    107.1611
```

sum

Erin L. Keller

The sum function calculates the sum of all values in the argument. If no arguments are given, the sum is “0” by definition. The input arguments can be numerical or complex vectors and the output will be an integer. If a non-numeric value is present in the argument, you will receive an error message.

```
sum(1:5)
```

```
## [1] 15
```

```
a<-1:5
sum(a+1)
```

```
## [1] 20
```

sweep()

Alex Burnham

Description:

Arguments:

- x = an array (matrix or higher dimension)
- FUN = finary opporator “-” subtraction symbol is default
- MARGIN 1 = rows and 2 = columns
- STATS = vecotor as long as either the row or column as sepcified by MARGIN
- check.margin = logical, if TRUE checks if margin equals dimensions of matrix x

Example:

```
# make a dataframe:
dataframe <- data.frame(cbind(c(2,4,6), c(1:3)))
print(dataframe)
```

```
##   X1 X2
## 1  2  1
## 2  4  2
## 3  6  3
```

```
# convert to a matrix
matrix <- data.matrix(frame=dataframe)

sweep(x=matrix, MARGIN = 2, STATS = c(2,3), check.margin=TRUE, FUN="-")
```

```
##      X1 X2
## [1,]  0 -2
## [2,]  2 -1
## [3,]  4  0
```

Explanation of Example: Returns a matrix where 2 and 3 are subtracted from each row

t

Erin L. Keller

t is a function that transposes a matrix or data frame (i.e. it will flip the column and rows). The input object, x, is typically a matrix or data frame; however, a coerced vector can be used as well. The ouput of t is a matrix with dim and dimnames constructed and all other elements of the original matrix.

```
a <- matrix(1:10, 2, 5)
colnames(a) <- LETTERS[1:5]
print(a)
```

```
##      A B C D  E
## [1,] 1 3 5 7  9
## [2,] 2 4 6 8 10
```

```
ta <- t(a)
print(ta)
```

```
##   [,1] [,2]
## A    1    2
## B    3    4
## C    5    6
## D    7    8
## E    9   10
```

tail()

Alex Burnham

Description: Shows the last rows of a data set (by default 6 but can be changed to any value)

Arguments:

- x = data set (data frame)
- n = number of rows displayed

Example:

```
# make a data frame:
dataframe <- data.frame(cbind(rep(c("blue", "red", "green"),3), c(1:9)))

# use tail
tail(x=dataframe, n=3)
```

```
##      X1 X2
## 7  blue  7
## 8   red  8
## 9 green  9
```

Explanation of Example: This shows me the last 3 rows of this data frame

tan

Peter Clark

One of the *Trig* fucntions, the **tan()** function computes the tangent value of numeric value. See Trig in help menu for a suite of other arguments Note: R always works with angles in radians, not in degrees.

Usage

tan(x) where x is a numeric value, array or vector

```
x <-pi
tan(x) # tangent value of pi
```

```
## [1] -1.224606e-16
```

```
tan(pi) # or just enter the value you desire
```

```
## [1] -1.224606e-16
```

```
tan(120*pi/180) # to calculate the cosine of an angle of 120 degrees, you must use pi, since R doe
sn't work in degrees
```

```
## [1] -1.732051
```

```
x <- c(pi, pi/4, 0)
tan(x) # combine values to recive multiple computations at once
```

```
## [1] -1.224606e-16  1.000000e+00  0.000000e+00
```

trunc()

Alexander Looi

This function simply cuts all the deciments off a numeric.

```
nums = runif(10, 0, 20)
trunc(nums)
```

```
##  [1]  5  5 18 11  1 14 14  5 10 16
```

```
# giving them inifinite or NA's returns the value entered.
trunc(Inf)
```

```
## [1] Inf
```

```
trunc(NA)
```

```
## [1] NA
```

```
trunc(-Inf)
```

```
## [1] -Inf
```

```
# a character value will not work
# trunc('a')
```

`union`

Matthias Nevins

`union()` is used on a **set** of vectors. union will join two vectors and discard any duplicated values.

```
unionX <- c(1:5)
unionX
```

```
## [1] 1 2 3 4 5
```

```
unionY <- c(3:8)
unionY
```

```
## [1] 3 4 5 6 7 8
```

```
union(unionX,unionY)
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
# The duplicates are removed and the set is now joined with union()
## More perfect??
```

`unlist`

Samantha Alger

unlist transforms a list structure into a vector.

```
# setting up a list
myList <- list(1:30,matrix(1:12,nrow=4,byrow=TRUE),
               LETTERS[1:5])
# view the list
print(myList)
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
##
## [[3]]
## [1] "A" "B" "C" "D" "E"
```

```
# using the unlist function
# output produces a vector containing all atomic components of the list
unlist(myList)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28"
## [29] "29" "30" "1"  "4"  "7"  "10" "2"  "5"  "8"  "11" "3"  "6"  "9"  "12"
## [43] "A"  "B"  "C"  "D"  "E"
```

var

April D. Makukhov

This function calculates the variance for vectors, matrices, or data frames. The main argument for this function is x, which specifies a numeric vector, matrix, or data frame. If comparing to another vector, matrix, or data frame, then y would be used for such an object. For this function, there is also a logical argument na.rm, which asks whether missing values should be removed (so can specify TRUE or FALSE with this argument).

```
var(1:10) # this provides the variance for a vector made up of values 1 through 10.
```

```
## [1] 9.166667
```

```
x<-c(2,3,5,7,9)
y<-c(1,4,2,6,2)
var(x,y) # finding the variance between vectors x and y
```

```
## [1] 1.5
```

warning()

Alexander Looi

This function allows you to track warning messages. These messages are not errors, since when warnings are thrown it generally means that the code exicuted successfuly, but something “bad” happened. Many functions can throw warnings, and usually the way to access these messages you must call the warning function. You can use the warning() object to write functions and have those functions throw warnings as well.

```
# typing:
warning()
```

```
## Warning:
```

```
# gives you warning messages thrown
```

which

Samantha Alger

Use which() to find a subset of data that meet a particular criteria

```
#Example 1:

#create a dataset
bird<- rep_len(c("finch","thrush","warbler"),20)

#use which to figure out the positions of a particular component
#of the dataset, which ones are "finch", which() returns the position.
which(bird =="finch")
```

```
## [1]  1  4  7 10 13 16 19
```

```
# Another example:

#For a given vector, which is the index of the 3rd non-NA value?
#create dataset
x <- c(1,NA,2,NA,3)

#which position is the third non-NA value?
which(!is.na(x))[3]
```

```
## [1] 5
```

with()

Alexander Looi

This function applies a function to an entire data set. You can then call the specific variables/columns you would like to be used by the funtion in the function itself.

```
R1 = runif(50, 1, 20)
R2 = runif(50, 1, 5)
R3 = runif(50, 33, 60)
categor = rep(letters[1:2], 25)

data = data.frame(R1, R2, R3, categor, stringsAsFactors = F)

# be careful, successful useage of this function is dependent on the function being applied to the
 dataset.
with(data, t.test(R1 ~ categor))
```

```
##
##  Welch Two Sample t-test
##
## data:  R1 by categor
## t = 0.98953, df = 44.999, p-value = 0.3277
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.717753  5.035763
## sample estimates:
## mean in group a mean in group b
##        11.03989         9.38088
```

```
# another example.
my_lm = with(data, lm(R1~R2))

# You can also apply multiple funtions as well
with(data, t.test(R1 ~ categor), plot(R1 ~ R3))
```

```
##
##  Welch Two Sample t-test
##
## data:  R1 by categor
## t = 0.98953, df = 44.999, p-value = 0.3277
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.717753  5.035763
## sample estimates:
## mean in group a mean in group b
##        11.03989         9.38088
```

`write.delim`

Peter Clark

In package *"caroline"*, this function allows you to write a tab delimited text file. This is a wrapper for *write.table* with the same options as *read.delim*. Files are saved to your working directory.

Usage

write.delim(df, file, quote = FALSE, row.names = FALSE, sep = "", …)

Arguments

- df: a dataframe.
    - file outputfile path.
    - quote should elements of the dataframe be quoted for output.
    - row.names should the output include rownames.
    - sep the delimiter between fields.
    - … other parameters passed to write.table.

```
library(caroline) # run package caroline
```

```
## Warning: package 'caroline' was built under R version 3.3.3
```

```
x <- data.frame(a = I("a \" quote"), b = pi) # assigns dataframe to variable x
write.delim(x, file = "mydata.tab") # writes dataframe for x to tab delimited file saved to your w
orking directory
```

`writeLines`

Matthias Nevins

At the most basic level, writeLines is used to write text lines from r code. The more advanced use of this function can write text to other types of connected files (text files, .html, etc.). See help(writeLines) for more details on advanced usage.

```
# For example, writeLines can be used with "LETTERS"
# We can assign LETTERS to variable B
b<-LETTERS
# Printing b gives us all the leters seperated by ""
head(b)
```

```
## [1] "A" "B" "C" "D" "E" "F"
```

```
writeLines(LETTERS) # writeLines applied to letters removes the "" separator
```

```
## A
## B
## C
## D
## E
## F
## G
## H
## I
## J
## K
## L
## M
## N
## O
## P
## Q
## R
## S
## T
## U
## V
## W
## X
## Y
## Z
```

```
# Another Example

a<-c("ant", "bee", "bug", "tree", "fern", "crow")
writeLines(a)
```

```
## ant
## bee
## bug
## tree
## fern
## crow
```

```
# To write to an actual file you can create a new text file and write to that file. See the exampl
e below using the file.create function
file.create("sample.txt")
```

```
## [1] TRUE
```

```
fileConn <- file("sample.txt")
writeLines(a, con = fileConn, sep = " ")
file.show("sample.txt")
```

write.csv

Samantha Alger

write.csv() is used to write data to a csv file

```
# a sample data frame

A <- 1:16
B <- rep(c("red","blue","green","NA"), each =4)
C <-runif(16)
data <- data.frame(A,B,C,stringsAsFactors = FALSE)

# write to a file, without row names
write.csv(data, "data.csv")

# same, except without row names
write.csv(data, "data.csv", row.names = FALSE)

# same, except of "NA", return blank cell
write.csv(data, "data.csv", row.names = FALSE, na = "")

#check out the csv file created.
read.csv("data.csv")
```

```
##     A     B          C
## 1   1   red 0.35794579
## 2   2   red 0.27888573
## 3   3   red 0.29521833
## 4   4   red 0.72553366
## 5   5  blue 0.52370814
## 6   6  blue 0.89870960
## 7   7  blue 0.55453434
## 8   8  blue 0.18094459
## 9   9 green 0.98767741
## 10 10 green 0.08449459
## 11 11 green 0.28015344
## 12 12 green 0.38677791
## 13 13  <NA> 0.66758145
## 14 14  <NA> 0.53247124
## 15 15  <NA> 0.23096133
## 16 16  <NA> 0.63227896
```

xor

April D. Makukhov

This command referred to as “exclusively OR”; it is used for when you want to make a statement in R where the outcome is one OR the other, but not both. For example, if you were flipping a coin, xor would be relevant to this situation because you can have heads or tails in a flip, but you can’t have both one or the other in a flip, it can only be one (so this is an ‘exclusive’ OR). This function is particularly useful for TRUE/FALSE statements (or 0s and 1s).

```
# creating x and y vectors made up of the same length of values, either 0 or 1 for each value
x<-c(1,0,0,1,1)
y<-c(0,0,1,0,1)
xor(x,y) #When we use xor, it takes the ones that are the same, either both 0 or 1 for x and y, an
d outputs a "FALSE", because the exclusive OR condition is not being satisfied. However, you'll no
tice the first, third, and fifth, values to output a "TRUE", because the values are different at t
hose positions in both vectors (either 0 or 1, but not both).
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```