

Handling massive data

Functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere “user” to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters. This interface provides an abstraction of the code to potential users. This abstraction simplifies the users’ lives because it relieves them from having to know every detail of how the code operates. In addition, the creation of an interface allows the developer to communicate to the user the aspects of the code that are important or are most relevant.

Functions in R

Functions in R are “first class objects”, which means that they can be treated much like any other R object.

Importantly,

- **Functions can be passed as arguments to other functions.**
- **Functions can be nested, so that you can define a function inside of another function**

Custom Functions

Function name

Function Arguments

```
bmi <- function(weight, height) {  
  height/100 -> height  
  height^2 -> height  
  return(weight/height)  
}
```

Code Block

Return value

```
> bmi(90,175)  
[1] 29.38776
```

```
> bmi(c(90,102), c(175,183))  
[1] 29.38776 30.45776
```

Your First Function

Functions are defined using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

Here's a simple function that takes no arguments and does nothing.

```
> f <- function() {  
+   ## This is an empty function  
+ }  
> ## Functions have their own class  
> class(f)  
[1] "function"  
> ## Execute this function  
> f()  
NULL
```

Here we create a function that actually has a non-trivial *function body*.

```
> f <- function() {  
+   cat("Hello, world!\n")  
+ }  
> f()  
Hello, world!
```

Your First Function

The last aspect of a basic function is the *function arguments*. These are the options that you can specify to the user that the user may explicitly set.

This next function returns the total number of characters printed to the console.

For this basic function, we can add an argument that determines how many times "Hello, world!" is printed to the console.

```
> f <- function(num) {  
+   for(i in seq_len(num)) {  
+     cat("Hello, world!\n")  
+   }  
+ }  
> f(3)  
Hello, world!  
Hello, world!  
Hello, world!
```

```
> f <- function(num) {  
+   hello <- "Hello, world!\n"  
+   for(i in seq_len(num)) {  
+     cat(hello)  
+   }  
+   chars <- nchar(hello) * num  
+   chars  
+ }  
> meaningoflife <- f(3)  
Hello, world!  
Hello, world!  
Hello, world!  
> print(meaningoflife)  
[1] 42
```

Your First Function

Here, for example, we could set the default value for num to be 1, so that if the function is called without the num argument being explicitly specified, then it will print “Hello, world!” to the console once.

```
> f <- function(num = 1) {  
+   hello <- "Hello, world!\n"  
+   for(i in seq_len(num)) {  
+     cat(hello)  
+   }  
+   chars <- nchar(hello) * num  
+   chars  
+ }  
> f() ## Use default value for 'num'  
Hello, world!  
[1] 14  
> f(2) ## Use user-specified value  
Hello, world!  
Hello, world!  
[1] 28
```

At this point, we have written a function that

- has one *formal argument* named num with a *default value* of 1. The *formal arguments* are the arguments included in the function definition.

The `formals()` function returns a list of all the formal arguments of a function

- prints the message “Hello, world!” to the console a number of times indicated by the argument num
- *returns* the number of characters printed to the console

Argument Matching

Calling an R function with arguments can be done in a variety of ways. This may be confusing at first, but it's really handy when doing interactive work at the command line. R functions arguments can be matched *positionally* or by name. Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc. So in the following call to `rnorm()`

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> mydata <- rnorm(100, 2, 1)      ## Generate some data
```

100 is assigned to the `n` argument, 2 is assigned to the `mean` argument, and 1 is assigned to the `sd` argument, all by positional matching.

Argument Matching

The following calls to the `sd()` function (which computes the empirical standard deviation of a vector of numbers) are all equivalent. Note that `sd()` has two arguments: `x` indicates the vector of numbers and `na.rm` is a logical indicating whether missing values should be removed or not.

```
> ## Positional match first argument, default for 'na.rm'  
> sd(mydata)  
[1] 1.014325  
> ## Specify 'x' argument by name, default for 'na.rm'  
> sd(x = mydata)  
[1] 1.014325  
> ## Specify both arguments by name  
> sd(x = mydata, na.rm = FALSE)  
[1] 1.014325
```

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.

In this example, the function `f()` has two arguments: `a` and `b`.

```
> f <- function(a, b) {  
+   a^2  
+ }  
> f(2)  
[1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the `2` gets positionally matched to `a`. This behavior can be good or bad. It's common to write a function that doesn't use an argument and not notice it simply because R never throws an error.

Lazy Evaluation

This example also shows lazy evaluation at work, but does eventually result in an error.

```
> f <- function(a, b) {  
+   print(a)  
+   print(b)  
+ }  
> f(45)  
[1] 45  
Error in print(b): argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because **b** did not have to be evaluated until after **print(a)**. Once the function tried to evaluate **print(b)** the function had to throw an error.

The ... Argument

There is a special argument in R known as the ... argument, which indicate a variable number of arguments that are usually passed on to other functions. The ... argument is often used when extending another function and you don't want to copy the entire argument list of the original function. For example, a custom plotting function may want to make use of the default plot() function along with its entire argument list. The function below changes the default for the type argument to the value type = "l" (the original default was type = "p").

```
myplot <- function(x, y, type = "l", ...) {  
  plot(x, y, type = type, ...)    ## Pass '...' to 'plot' function  
}
```

Generic functions use ... so that extra arguments can be passed to methods.

```
> mean  
function (x, ...)  
  UseMethod("mean")  
<bytecode: 0x7fe2e9ae40>  
<environment: namespace:base>
```

Summary

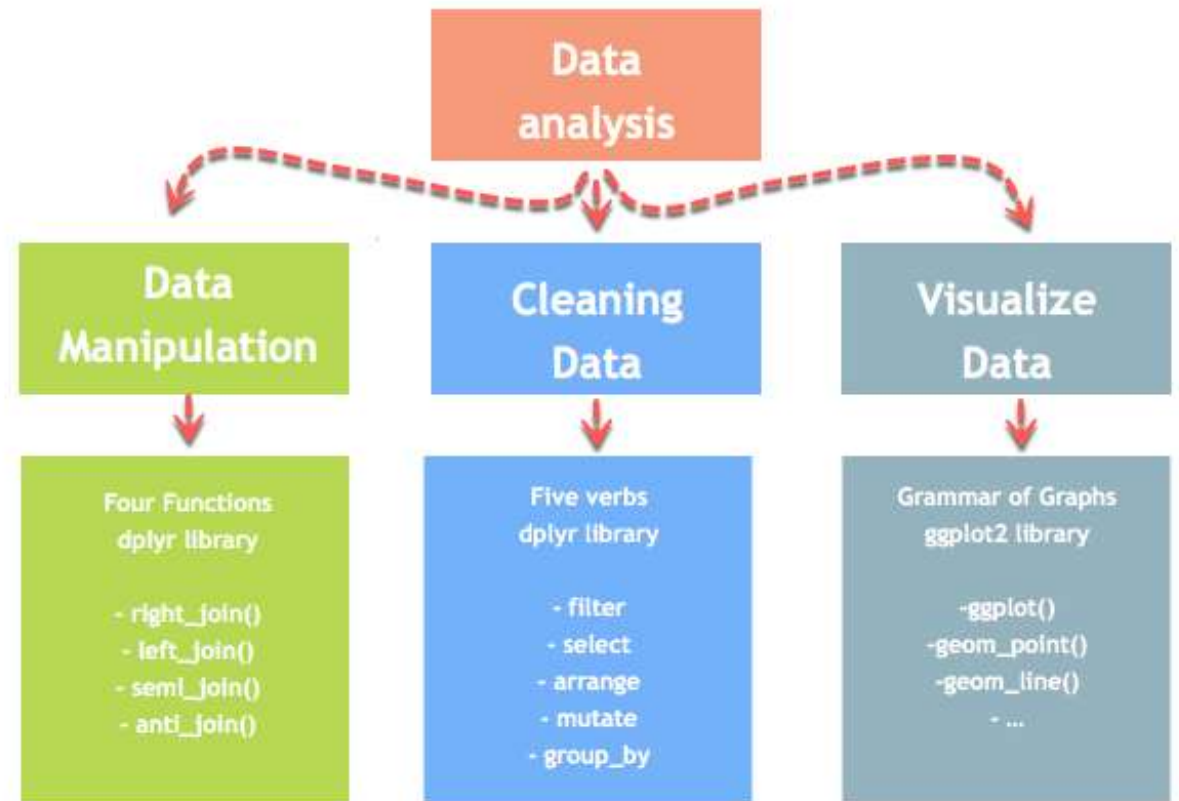
- Functions can be defined using the `function()` directive and are assigned to R objects just like any other R object
- Functions can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special `...` argument in a function definition.

Introduction to Data Analysis

Data analysis can be divided into three parts:

- Extraction:** First, we need to collect the data from many sources and combine them.
- Transform:** This step involves the data manipulation. Once we have consolidated all the sources of data, we can begin to clean the data.
- Visualize:** The last move is to visualize our data to check irregularity.

One of the most significant challenges faced by data scientists is the data manipulation. Data is never available in the desired format. Data scientists need to spend at least half of their time, cleaning and manipulating the data. That is one of the most critical assignments in the job. If the data manipulation process is not complete, precise and rigorous, the model will not perform correctly.



Tidyverse Packages



- Tibble - data storage



- ReadR - reading data from files



- TidyR - Model data correctly



- DplyR - Manipulate and filter data



- Ggplot2 - Draw figures and graphs

The dplyr Package

The dplyr package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package. **The dplyr package does not provide any “new” functionality to R** per se, in the sense that everything dplyr does could already be done with base R, but it *greatly simplifies existing functionality* in R.

One important contribution of the dplyr package is that it provides a **“grammar”** (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar). This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the dplyr functions are **very fast**, as many key operations are coded in C++.

Loop Functions

Writing for and while loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- **lapply(): Loop over a list and evaluate a function on each element**
- **sapply(): Same as lapply but try to simplify the result**
- **apply(): Apply a function over the margins of an array**
- **tapply(): Apply a function over subsets of a vector**
- **mapply(): Multivariate version of lapply**

An auxiliary function split is also useful, particularly in conjunction with lapply.

apply()

The `apply()` function is used to evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of an array of matrices. Using `apply()` is not really faster than writing a loop, but it works in one line and is highly compact.

```
> str(apply)  
function (X, MARGIN, FUN, ...)
```

The arguments to `apply()` are

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be “retained”.
- `FUN` is a function to be applied
- `...` is for other arguments to be passed to `FUN`

apply()

The simplest example is to sum a matrix over all the columns. The code `apply(m1, 2, sum)` will apply the sum function to the matrix 5×6 and return the sum of each column accessible in the dataset.

```
m1 <- matrix(C<-(1:10),nrow=5, ncol=6)
m1
a_m1 <- apply(m1, 2, sum)
a_m1
```

Output:

```
> m1
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6    1    6    1    6
[2,]    2    7    2    7    2    7
[3,]    3    8    3    8    3    8
[4,]    4    9    4    9    4    9
[5,]    5   10    5   10    5   10
> a_m1 <- apply(m1, 2, sum)
> a_m1
[1] 15 40 15 40 15 40
>
```

sum of column

Best practice: Store the values before printing it to the console.

apply()

Here I create a 20 by 10 matrix of Normal random numbers. I then compute the mean of each column.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean) ## Take the mean of each column
[1] 0.02218266 -0.15932850 0.09021391 0.14723035 -
0.22431309 -0.49657847
[7] 0.30095015 0.07703985 -0.20818099 0.06809774
```

I can also compute the sum of each row.

```
> apply(x, 1, sum) ## Take the mean of each row
[1] -0.48483448 5.33222301 -3.33862932 -1.39998450
2.37859098 0.01082604
[7] -6.29457190 -0.26287700 0.71133578 -3.38125293 -
4.67522818 3.01900232
[13] -2.39466347 -2.16004389 5.33063755 -2.92024635
3.52026401 -1.84880901
[19] -4.10213912 5.30667310
```

Note that in both calls to `apply()`, the return value was a vector of numbers.

apply()

You've probably noticed that the second argument is either a 1 or a 2, depending on whether we want row statistics or column statistics. What exactly *is* the second argument to `apply()`?

The `MARGIN` argument essentially indicates to `apply()` which dimension of the array you want to preserve or retain. So when taking the mean of each column, I specify

```
> apply(x, 2, mean)
```

because I want to collapse the first dimension (the rows) by taking the mean and I want to preserve the number of columns. Similarly, when I want the row sums, I run

```
> apply(x, 1, sum)
```

because I want to collapse the columns (the second dimension) and preserve the number of rows (the first dimension).

Col/Row Sums and Means

For the special case of column/row sums and column/row means of matrices, we have some useful shortcuts.

- `rowSums = apply(x, 1, sum)`

- `rowMeans = apply(x, 1, mean)`

- `colSums = apply(x, 2, sum)`

- `colMeans = apply(x, 2, mean)`

The shortcut functions are heavily optimized and hence are *much* faster, but you probably won't notice unless you're using a large matrix. Another nice aspect of these functions is that they are a bit more descriptive. It's arguably more clear to write `colMeans(x)` in your code than `apply(x, 2, mean)`.

lapply()

The `lapply()` function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the `l` is for "list").

This function takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list()`.

The body of the `lapply()` function can be seen here.

```
> lapply
function (X, FUN, ...)
{
  FUN <- match.fun(FUN)
  if (!is.vector(X) || is.object(X))
    X <- as.list(X)
  .Internal(lapply(X, FUN))
}
<bytecode: 0x7ff2f68331c0>
<environment: namespace:base>
```

Note that the actual looping is done internally in C code for efficiency reasons.
It's important to remember that `lapply()` always returns a list, regardless of the class of the input.

lapply()

A very easy example can be to change the string value of a matrix to lower case with to lower function. We construct a matrix with the name of the famous movies. The name is in upper case format.

```
movies <- c("SPYDERMAN","BATMAN","VERTIGO","CHINATOWN")  
movies_lower <- lapply(movies, tolower)  
str(movies_lower)
```

We can use `unlist()` to convert the list into a vector.

```
movies_lower <- unlist(lapply(movies,tolower))  
str(movies_lower)
```

Output:

```
## chr [1:4] "spyderman" "batman" "vertigo" "chinatown"
```


lapply()

Here's an example of applying the `mean()` function to all elements of a list. If the original list has names, the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))  
> lapply(x, mean)  
$a  
[1] 3  
  
$b  
[1] 0.1322028
```

Notice that here we are passing the `mean()` function as an argument to the `lapply()` function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses `()` like you do when you are *calling* a function.

lapply()

Here is another example of using lapply().

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =  
rnorm(100, 5))  
> lapply(x, mean)  
$a  
[1] 2.5  
  
$b  
[1] 0.248845  
  
$c  
[1] 0.9935285  
  
$d  
[1] 5.051388
```

You can use **lapply()** to evaluate a function multiple times each with a different argument.

lapply()

Below, is an example where I call the `runif()` function (to generate uniformly distributed random variables) four times, each time generating a different number of random numbers.

```
> x <- 1:4  
> lapply(x, runif)  
[[1]]  
[1] 0.02778712  
  
[[2]]  
[1] 0.5273108 0.8803191  
  
[[3]]  
[1] 0.37306337 0.04795913 0.13862825  
  
[[4]]  
[1] 0.3214921 0.1548316 0.1322282 0.2213059
```

lapply()

When you pass a function to `lapply()`, `lapply()` takes elements of the list and passes them as the *first argument* of the function you are applying. In the above example, the first argument of `runif()` is `n`, and so the elements of the sequence `1:4` all got passed to the `n` argument of `runif()`.

Functions that you pass to `lapply()` may have other arguments. For example, the `runif()` function has a `min` and `max` argument too. In the example above I used the default values for `min` and `max`. How would you be able to specify different values for that in the context of `lapply()`?

Here is where the `...` argument to `lapply()` comes into play. Any arguments that you place in the `...` argument will get passed down to the function being applied to the elements of the list.

Here, the `min = 0` and `max = 10` arguments are passed down to `runif()` every time it gets called.

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 2.263808

[[2]]
[1] 1.314165 9.815635

[[3]]
[1] 3.270137 5.069395 6.814425

[[4]]
[1] 0.9916910 1.1890256 0.5043966 9.2925392
```

lapply()

The `lapply()` function and its friends make heavy use of *anonymous* functions. Anonymous functions have no names. These are functions are generated “on the fly” as you are using `lapply()`. Once the call to `lapply()` is finished, the function disappears and does not appear in the workspace.

Here I am creating a list that contains two matrices.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
  [,1] [,2]
[1,]  1  3
[2,]  2  4

$b
  [,1] [,2]
[1,]  1  4
[2,]  2  5
[3,]  3  6
```

lapply()

Suppose I wanted to extract the first column of each matrix in the list. I could write an anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) { elt[,1] })  
$a  
[1] 1 2  
$b  
[1] 1 2 3
```

Notice that I put the function() definition right in the call to lapply(). This is perfectly legal and acceptable. You can put an arbitrarily complicated function definition inside lapply(), but if it's going to be more complicated, it's probably a better idea to define the function separately. For example, I could have done the following.

```
> f <- function(elt) {  
+   elt[, 1]  
+ }  
> lapply(x, f)  
$a  
[1] 1 2  
$b  
[1] 1 2 3
```

Now the function is no longer anonymous; it's name is **f**. Whether you use an anonymous function or you define a function first depends on your context. If you think the function **f** is something you're going to need a lot in other parts of your code, you might want to define it separately. But if you're just going to use it for this call to **lapply()**, then it's probably simpler to use an anonymous function.

sapply()

The `sapply()` function behaves similarly to `lapply()`; the only real difference is in the return value. `sapply()` will try to simplify the result of `lapply()` if possible.

Essentially, `sapply()` calls `lapply()` on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

supply()

Here's the result of calling lapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =  
rnorm(100, 5))  
> lapply(x, mean)  
$a  
[1] 2.5  
  
$b  
[1] -0.251483  
  
$c  
[1] 1.481246  
  
$d  
[1] 4.968715
```


sapply()

Here's the result of calling `lapply()`

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =  
rnorm(100, 5))  
> lapply(x, mean)  
$a  
[1] 2.5  
  
$b  
[1] -0.251483  
  
$c  
[1] 1.481246  
  
$d  
[1] 4.968715
```

Here's the result of calling `sapply()` on the same list.

```
> sapply(x, mean)  
      a      b      c      d  
2.500000 -0.251483 1.481246 4.968715
```

Because the result of `lapply()` was a list where each element had length 1, `sapply()` collapsed the output into a numeric vector, which is often more useful than a list.

tapply

`tapply()` is used to apply a function over subsets of a vector. It can be thought of as a combination of `split()` and `sapply()` for vectors only.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

the arguments to `tapply()` are as follows:

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

tapply

Given a vector of numbers, one simple operation is to take group means.

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
      1      2      3
0.1896235 0.5336667 0.9568236
```

tapply

We can also take the group means without simplifying the result, which will give us a list. For functions that return a single value, usually, this is not what we want, but it can be done.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1896235
$`2`
[1] 0.5336667
$`3`
[1] 0.9568236
```

We can also apply functions that return more than a single value. In this case, **tapply()** will not simplify the result and will return a list. Here's an example of finding the range of each sub-group.

```
> tapply(x, f, range)
$`1`
[1] -1.869789  1.497041
$`2`
[1] 0.09515213 0.86723879
$`3`
[1] -0.5690822  2.3644349
```

mapply()

The `mapply()` function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that `lapply()` and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what `mapply()` is for.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

The arguments to `mapply()` are

- `FUN` is a function to apply
- `...` contains R objects to apply over
- `MoreArgs` is a list of other arguments to `FUN`.
- `SIMPLIFY` indicates whether the result should be simplified

The `mapply()` function has a different argument order from `lapply()` because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the `...` argument because we can apply over an arbitrary number of R objects.

mapply()

For example, the following is tedious to type
`list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))`
With `mapply()`, instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

rep: Replicate Elements of Vectors and Lists

This passes the sequence `1:4` to the first argument of `rep()` and the sequence `4:1` to the second argument.

mapply()

Here's another example for simulating random Normal variables.

```
> noise <- function(n, mean, sd) {  
+   rnorm(n, mean, sd)  
+ }  
> ## Simulate 5 random numbers  
> noise(5, 1, 2)  
[1] -0.5196913  3.2979182 -0.6849525  1.7828267  
2.7827545  
>  
> ## This only simulates 1 set of numbers, not 5  
> noise(1:5, 1:5, 2)  
[1] -1.670517  2.796247  2.776826  5.351488  3.422804
```

mapply()

Here's another example for simulating random Normal variables.

```
> noise <- function(n, mean, sd) {  
+   rnorm(n, mean, sd)  
+ }  
> ## Simulate 5 random numbers  
> noise(5, 1, 2)  
[1] -0.5196913  3.2979182 -0.6849525  1.7828267  
2.7827545  
>  
> ## This only simulates 1 set of numbers, not 5  
> noise(1:5, 1:5, 2)  
[1] -1.670517  2.796247  2.776826  5.351488  3.422804
```

we can use `mapply()` to pass the sequence `1:5` separately to the `noise()` function so that we can get 5 sets of random numbers, each with a different length and mean.

```
> mapply(noise, 1:5, 1:5, 2)  
[[1]]  
[1] 0.8260273  
  
[[2]]  
[1] 4.764568 2.336980  
  
[[3]]  
[1] 4.6463819 2.5582108 0.9412167  
  
[[4]]  
[1] 3.978149 1.550018 -1.192223 6.338245  
  
[[5]]  
[1] 2.826182 1.347834 6.990564 4.976276 3.800743
```


split()

The `split()` function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The arguments to `split()` are

```
> str(split)  
function (x, f, drop = FALSE, ...)
```

where

- `x` is a vector (or list) or data frame
- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped

The combination of `split()` and a function like `lapply()` or `sapply()` is a common paradigm in R. The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets. The results of applying that function over the subsets are then collated and returned as an object. This sequence of operations is sometimes referred to as “map-reduce” in other contexts.

split()

Here we simulate some data and split it according to a factor variable. Note that we use the `gl()` function to “generate levels” in a factor variable.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
[1] 0.3981302 -0.4075286 1.3242586 -0.7012317 -0.5806143 -1.0010722
[7] -0.6681786 0.9451850 0.4337021 1.0051592

$`2`
[1] 0.34822440 0.94893818 0.64667919 0.03527777 0.59644846 0.41531800
[7] 0.07689704 0.52804888 0.96233331 0.70874005

$`3`
[1] 1.13444766 1.76559900 1.95513668 0.94943430 0.69418458 1.89367370
[7] -0.04729815 2.97133739 0.61636789 2.65414530
```

split()

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.07478098

$`2`
[1] 0.5266905

$`3`
[1] 1.458703
```

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form
- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.
- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere
- The `split()` function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.

dplyr Grammar

Some of the key “verbs” provided by the dplyr package are

- **select**: return a subset of the columns of a data frame, using a flexible notation
- **filter**: extract a subset of rows from a data frame based on logical conditions
- **arrange**: reorder rows of a data frame
- **rename**: rename variables in a data frame
- **mutate**: add new variables/columns or transform existing variables
- **summarise / summarize**: generate summary statistics of different variables in the data frame, possibly within strata
- **%>%**: the “pipe” operator is used to connect multiple verb actions together into a pipeline

The dplyr package has a number of its own data types that it takes advantage of. For example, there is a handy print method that prevents you from printing a lot of data to the console. Most of the time, these additional data types are transparent to the user and do not need to be worried about.

Common dplyr Function Properties

All of the functions that we will discuss will have a few common characteristics. In particular,

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the `$` operator (just use the column names).
3. The return result of a function is a new data frame
4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be [tidy](#). In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

```
> install.packages("dplyr")  
> library(dplyr)
```

select()

For the examples we will be using a dataset containing air pollution and temperature data for the [city of Chicago](#) in the U.S.

```
> chicago <- read_csv("chicago.rds")
> dim(chicago)
[1] 6940  8
> str(chicago)
'data.frame':      6940 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd      : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
 $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
 $ date      : Date, format: "1987-01-01" "1987-01-02" ...
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

You can see some basic characteristics of the dataset with the `dim()` and `str()` functions.

select()

The `select()` function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but any *given* analysis might only use a subset of variables or observations. The `select()` function allows you to get the few columns you might need. Suppose we wanted to take the first 3 columns only. There are a few ways to do this. We could for example use numerical indices. But we can also use the names directly.

```
> names(chicago)[1:3]
[1] "city" "tmpd" "dptp"
> subset <- select(chicago, city:dptp)
> head(subset)
  city tmpd  dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
4 chic 29.0 28.625
5 chic 32.0 28.875
6 chic 40.0 35.125
```

Note that the `:` normally cannot be used with names or strings, but inside the `select()` function you can use it to specify a range of variable names.

select()

You can also *omit* variables using the `select()` function by using the negative sign. With `select()` you can do

```
> select(chicago, -(city:dptp))
```

which indicates that we should include every variable *except* the variables `city` through `dptp`. The equivalent code in base R would be

```
> i <- match("city", names(chicago))  
> j <- match("dptp", names(chicago))  
> head(chicago[, -(i:j)])
```

Not super intuitive, right?

select()

The `select()` function also allows a special syntax that allows you to specify variable names based on patterns. So, for example, if you wanted to keep every variable that ends with a "2", we could do

```
> subset <- select(chicago, ends_with("2"))
> str(subset)
'data.frame':      6940 obs. of  4 variables:
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Or if we wanted to keep every variable that starts with a "d", we could do

```
> subset <- select(chicago, starts_with("d"))
> str(subset)
'data.frame':      6940 obs. of  2 variables:
 $ dptp: num  31.5 29.9 27.4 28.6 28.9 ...
 $ date: Date, format: "1987-01-01" "1987-01-02" ...
```

filter()

The `filter()` function is used to extract subsets of **rows** from a data frame. This function is similar to the existing `subset()` function in R but is quite a bit faster. Suppose we wanted to extract the rows of the `chicago` data frame where the levels of PM2.5 are greater than 30 (which is a reasonably high level), we could do

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> str(chic.f)
'data.frame':      194 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd      : num  23 28 55 59 57 57 75 61 73 78 ...
 $ dptp      : num  21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
 $ date      : Date, format: "1998-01-17" "1998-01-23" ...
 $ pm25tmean2: num  38.1 34 39.4 35.4 33.3 ...
 $ pm10tmean2: num  32.5 38.7 34 28.5 35 ...
 $ o3tmean2  : num  3.18 1.75 10.79 14.3 20.66 ...
 $ no2tmean2 : num  25.3 29.4 25.3 31.4 26.8 ...
```

You can see that there are now only 194 rows in the data frame and the distribution of the `pm25tmean2` values is.

```
> summary(chic.f$pm25tmean2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 30.05  32.12  35.04  36.63  39.53  61.50
```

filter()

We can place an arbitrarily complex logical sequence inside of `filter()`, so we could for example extract the rows where PM2.5 is greater than 30 *and* temperature is greater than 80 degrees Fahrenheit.

```
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
```

```
> select(chic.f, date, tmpd, pm25tmean2)
```

	date	tmpd	pm25tmean2
1	1998-08-23	81	39.60000
2	1998-09-06	81	31.50000
3	2001-07-20	82	32.30000
4	2001-08-01	84	43.70000
5	2001-08-08	85	38.83750
6	2001-08-09	84	38.20000
7	2002-06-20	82	33.00000
8	2002-06-23	82	42.50000
9	2002-07-08	81	33.10000
10	2002-07-18	82	38.85000
11	2003-06-25	82	33.90000
12	2003-07-04	84	32.90000
13	2005-06-24	86	31.85714
14	2005-06-27	82	51.53750
15	2005-06-28	85	31.20000
16	2005-07-17	84	32.70000
17	2005-08-03	84	37.90000

Now there are only 17 observations where both of those conditions are met.

arrange()

The `arrange()` function is used to reorder rows of a data frame according to one of the variables/columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R. The `arrange()` function simplifies the process quite a bit. Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
> chicago <- arrange(chicago, date)
```

We can now check the first few rows

```
> head(select(chicago, date, pm25tmean2), 3)
  date pm25tmean2
1 1987-01-01      NA
2 1987-01-02      NA
3 1987-01-03      NA
```

and the last few rows.

```
> tail(select(chicago, date, pm25tmean2), 3)
  date pm25tmean2
6938 2005-12-29  7.45000
6939 2005-12-30 15.05714
6940 2005-12-31 15.00000
```

Columns can be arranged in descending order too by using the special `desc()` operator.

```
> chicago <- arrange(chicago, desc(date))
```

arrange()

Looking at the first three and last three rows shows the dates in descending order.

```
> head(select(chicago, date, pm25tmean2), 3)
  date pm25tmean2
1 2005-12-31  15.00000
2 2005-12-30  15.05714
3 2005-12-29   7.45000
> tail(select(chicago, date, pm25tmean2), 3)
  date pm25tmean2
6938 1987-01-03    NA
6939 1987-01-02    NA
6940 1987-01-01    NA
```

rename()

The dew point is **the temperature below which the water vapour in a volume of air at a constant pressure will condense into liquid water**. It is the temperature at which the air is saturated with moisture.

Renaming a variable in a data frame in R is surprisingly hard to do! The `rename()` function is designed to make this process easier. Here you can see the names of the first five variables in the `chicago` data frame.

```
> head(chicago[, 1:5], 3)
  city tmpd dptp    date pm25tmean2
1 chic  35 30.1 2005-12-31  15.00000
2 chic  36 31.0 2005-12-30  15.05714
3 chic  35 29.4 2005-12-29   7.45000
```

The `dptp` column is supposed to represent the dew point temperature and the `pm25tmean2` column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible.

```
> chicago <- rename(chicago, dewpoint = dptp, pm25 =
pm25tmean2)
> head(chicago[, 1:5], 3)
  city tmpd dewpoint    date  pm25
1 chic  35   30.1 2005-12-31 15.00000
2 chic  36   31.0 2005-12-30 15.05714
3 chic  35   29.4 2005-12-29  7.45000
```

The syntax inside the `rename()` function is to have the new name on the left-hand side of the `=` sign and the old name on the right-hand side.

mutate()

The `mutate()` function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing variables and `mutate()` provides a clean interface for doing that.

For example, with air pollution data, we often want to *detrend* the data by subtracting the mean from the data. That way we can look at whether a given day's air pollution level is higher than or less than average (as opposed to looking at its absolute level).

Here we create a `pm25detrend` variable that subtracts the mean from the `pm25` variable.

```
> chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
> head(chicago)
```

	city	tmpd	dewpoint	date	pm25	pm10tmean2	o3tmean2	no2tmean2
1	chic	35	30.1	2005-12-31	15.00000	23.5	2.531250	13.25000
2	chic	36	31.0	2005-12-30	15.05714	19.2	3.034420	22.80556
3	chic	35	29.4	2005-12-29	7.45000	23.5	6.794837	19.97222
4	chic	37	34.5	2005-12-28	17.75000	27.5	3.260417	19.28563
5	chic	40	33.6	2005-12-27	23.56000	27.0	4.468750	23.50000
6	chic	35	29.6	2005-12-26	8.40000	8.5	14.041667	16.81944

```
pm25detrend
1 -1.230958
2 -1.173815
3 -8.780958
4  1.519042
5  7.329042
6 -7.830958
```


transmute()

There is also the related `transmute()` function, which does the same thing as `mutate()` but then *drops all non-transformed variables*. Here we detrend the PM10 and ozone (O3) variables.

```
> head(transmute(chicago,
+               pm10detrend = pm10tmean2 - mean(pm10tmean2,
na.rm = TRUE),
+               o3detrend = o3tmean2 - mean(o3tmean2, na.rm =
TRUE)))
  pm10detrend o3detrend
1 -10.395206 -16.904263
2 -14.695206 -16.401093
3 -10.395206 -12.640676
4  -6.395206 -16.175096
5  -6.895206 -14.966763
6 -25.395206  -5.393846
```

Note that there are only two columns in the transmuted data frame.

group_by()

The `group_by()` function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the date variable. In conjunction with the `group_by()` function we often use the `summarize()` function (or `summarise()` for some parts of the world).

The general operation here is a combination of splitting a data frame into separate pieces defined by a variable or group of variables (`group_by()`), and then applying a summary function across those subsets (`summarize()`).

First, we can create a year variable using `as.POSIXlt()`.

```
> chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

Now we can create a separate data frame that splits the original data frame by year.

```
> years <- group_by(chicago, year)
```

Date-time Conversion Functions

Functions to manipulate objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times.

group_by()

Finally, we compute summary statistics for each year in the data frame with the `summarize()` function.

```
> summarize(years, pm25 = mean(pm25, na.rm = TRUE),  
+           o3 = max(o3tmean2, na.rm = TRUE),  
+           no2 = median(no2tmean2, na.rm = TRUE))  
`summarise()` ungrouping output (override with `.groups` argument)  
# A tibble: 19 x 4  
  year pm25  o3  no2  
  <dbl> <dbl> <dbl> <dbl>  
1 1987 NaN  63.0 23.5  
2 1988 NaN  61.7 24.5  
3 1989 NaN  59.7 26.1  
4 1990 NaN  52.2 22.6  
5 1991 NaN  63.1 21.4  
6 1992 NaN  50.8 24.8  
7 1993 NaN  44.3 25.8  
8 1994 NaN  52.2 28.5  
9 1995 NaN  66.6 27.3  
10 1996 NaN  58.4 26.4  
11 1997 NaN  56.5 25.5  
12 1998 18.3 50.7 24.6  
13 1999 18.5 57.5 24.7  
14 2000 16.9 55.8 23.5  
15 2001 16.9 51.8 25.1  
16 2002 15.3 54.9 22.7  
17 2003 15.2 56.2 24.6  
18 2004 14.6 44.5 23.4  
19 2005 16.2 58.8 22.6
```

`summarize()` returns a data frame with `year` as the first column, and then the annual averages of `pm25`, `o3`, and `no2`.

group_by()

In a slightly more complicated example, we might want to know what are the average levels of ozone (o3) and nitrogen dioxide (no2) within quintiles of pm25. A slicker way to do this would be through a regression model, but we can actually do this quickly with `group_by()` and `summarize()`.

First, we can create a categorical variable of pm25 divided into quintiles.

```
> qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
> chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))
```

Now we can group the data frame by the `pm25.quint` variable.

```
> quint <- group_by(chicago, pm25.quint)
```

Finally, we can compute the mean of o3 and no2 within quintiles of pm25.

```
> summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),
+           no2 = mean(no2tmean2, na.rm = TRUE))
# A tibble: 6 x 3
  pm25.quint  o3  no2
  <fct>      <dbl> <dbl>
1 (1.7,8.7]  21.7  18.0
2 (8.7,12.4] 20.4  22.1
3 (12.4,16.7] 20.7  24.4
4 (16.7,22.6] 19.9  27.3
5 (22.6,61.5] 20.3  29.6
6 <NA>      18.8  25.8
```

From the table, it seems there isn't a strong relationship between pm25 and o3, but there appears to be a positive correlation between pm25 and no2. More sophisticated statistical modeling can help to provide precise answers to these questions, but a simple application of dplyr functions can often get you most of the way there.


%>%

The pipeline operator %>% is very handy for stringing together multiple **dplyr** functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

```
> third(second(first(x)))
```

This nesting is not a natural way to think about a sequence of operations. The %>% operator allows you to string operations in a left-to-right fashion, i.e.

```
> first(x) %>% second %>% third
```



%>%

Take the example that we just did in the last section where we computed the mean of `o3` and `no2` within quintiles of `pm25`.

There we had to

1.create a new variable `pm25.quint`

2.split the data frame by that new variable

3.compute the mean of `o3` and `no2` in the sub-groups defined by `pm25.quint`

That can be done with the following sequence in a single R expression.

This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls.

```
> mutate(chicago, pm25.quint = cut(pm25, qq)) %>%  
+   group_by(pm25.quint) %>%  
+   summarize(o3 = mean(o3tmean2, na.rm = TRUE),  
+             no2 = mean(no2tmean2, na.rm = TRUE))  
`summarise()` ungrouping output (override with `.groups`  
argument)  
# A tibble: 6 x 3  
  pm25.quint    o3    no2  
  <fct>      <dbl> <dbl>  
1 (1.7,8.7]    21.7  18.0  
2 (8.7,12.4]   20.4  22.1  
3 (12.4,16.7]  20.7  24.4  
4 (16.7,22.6]  19.9  27.3  
5 (22.6,61.5]  20.3  29.6  
6 <NA>         18.8  25.8
```

%>%

Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data.

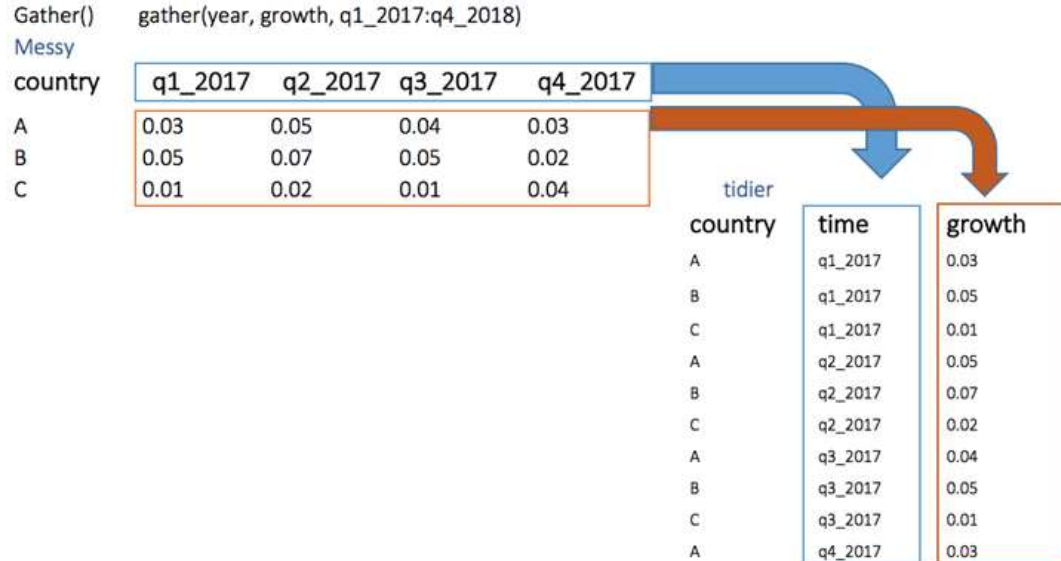
Here we can see that **o3** tends to be low in the winter months and high in the summer while **no2** is higher in the winter and lower in the summer.

```
> mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
+   group_by(month) %>%
+   summarize(pm25 = mean(pm25, na.rm = TRUE),
+             o3 = max(o3tmean2, na.rm = TRUE),
+             no2 = median(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups`
argument)
# A tibble: 12 x 4
  month pm25  o3  no2
  <dbl> <dbl> <dbl> <dbl>
1     1  17.8 28.2 25.4
2     2  20.4 37.4 26.8
3     3  17.4 39.0 26.8
4     4  13.9 47.9 25.0
5     5  14.1 52.8 24.2
6     6  15.9 66.6 25.0
7     7  16.6 59.5 22.4
8     8  16.9 54.0 23.0
9     9  15.9 57.5 24.5
10    10  14.2 47.1 24.2
11    11  15.2 29.5 23.6
12    12  17.5 27.7 24.5
```

gather()

The objectives of the gather() function is to transform the data from wide to long.

Below, we can visualize the concept of reshaping wide to long. We want to create a single column named growth, filled by the rows of the quarter variables.



```
library(tidyr) # Create a messy dataset
messy <- data.frame( country = c("A", "B", "C"), q1_2017 =
c(0.03, 0.05, 0.01), q2_2017 = c(0.05, 0.07, 0.02), q3_2017 =
c(0.04, 0.05, 0.01), q4_2017 = c(0.03, 0.02, 0.04))
# Reshape the data
tidier <- messy %>% gather(quarter, growth, q1_2017:q4_2017)
tidier
```

Output:

```
## country quarter growth
## 1 A q1_2017 0.03
## 2 B q1_2017 0.05
## 3 C q1_2017 0.01
## 4 A q2_2017 0.05
## 5 B q2_2017 0.07
## 6 C q2_2017 0.02
## 7 A q3_2017 0.04
## 8 B q3_2017 0.05
## 9 C q3_2017 0.01
## 10 A q4_2017 0.03
## 11 B q4_2017 0.02
## 12 C q4_2017 0.04
```


spread()

The spread() function does the opposite of gather.

We can reshape the tidier dataset back to messy with spread()

```
# Reshape the data
messy_1 <- tidier %>%
  spread(quarter, growth)
messy_1
```

Output:

```
## country q1_2017 q2_2017 q3_2017 q4_2017
## 1 A 0.03 0.05 0.04 0.03
## 2 B 0.05 0.07 0.05 0.02
## 3 C 0.01 0.02 0.01 0.04
```

separate()

The `separate()` function splits a column into two according to a separator. This function is helpful in some situations where the variable is a date. Our analysis can require focussing on month and year and we want to separate the column into two new variables.

We can split the quarter from the year in the tidier dataset by applying the `separate()` function.

```
separate_tidier <- tidier %>%  
  separate(quarter, c("Qrt", "year"), sep = "_")  
head(separate_tidier)
```

Output:

```
## country Qrt year growth  
## 1 A q1 2017 0.03  
## 2 B q1 2017 0.05  
## 3 C q1 2017 0.01  
## 4 A q2 2017 0.05  
## 5 B q2 2017 0.07  
## 6 C q2 2017 0.02
```

unite()

The unite() function concatenates two columns into one.

In the above example, we separated quarter from year. What if we want to merge them. We use the following code:

```
unit_tidier <- separate_tidier %>%  
  unite(Quarter, Qrt, year, sep = "_")  
head(unit_tidier)
```

Output:

```
## country Quarter growth  
## 1 A q1_2017 0.03  
## 2 B q1_2017 0.05  
## 3 C q1_2017 0.01  
## 4 A q2_2017 0.05  
## 5 B q2_2017 0.07  
## 6 C q2_2017 0.02
```

Summary

The dplyr package provides a concise set of operations for managing data frames. With these functions we can do a number of complex operations in just a few lines of code. In particular, we can often conduct the beginnings of an exploratory analysis with the powerful combination of `group_by()` and `summarize()`.

Once you learn the dplyr grammar there are a few additional benefits

- dplyr can work with other data frame “backends” such as SQL databases. There is an SQL interface for relational databases via the DBI package
- dplyr can be integrated with the `data.table` package for large fast tables

The dplyr package is handy way to both simplify and speed up your data frame management code. It’s rare that you get such a combination at the same time!

Data Frames

The *data frame* is a key data structure in statistics and in R. The basic structure of a data frame is that there is **one observation per row and each column represents a variable, a measure, feature, or characteristic of that observation**. R has an internal implementation of data frames that is likely the one you will use most often. However, there are packages on CRAN that implement data frames via things like relational databases that allow you to operate on very very large data frames (but we won't discuss them here).

Given the importance of managing data frames, it's important that we have good tools for dealing with them. In previous chapters we have already discussed some tools like the `subset()` function and the use of `[]` and `$` operators to extract subsets of data frames. However, other operations, like **filtering, re-ordering, and collapsing, can often be tedious operations in R** whose syntax is not very intuitive. The `dplyr` package is designed to mitigate a lot of these problems and to provide a highly optimized set of routines specifically for dealing with data frames.

Data Frame

- Collection of vectors with same lengths
- Gain the concept of 'rows'

```
all.results$mon  
mean(all.results$mon)
```

all.results				
	"mon"	"tue"	"wed"	"pass"
	1	2	3	4
1	0.8	0.9	0.8	T
2	0.6	0.7	0.5	F
3	0.2	0.3	0.3	F
4	0.8	0.8	0.9	T
5	0.6	1.0	0.9	T

Tibble

- Collection of vectors with same lengths
- Gain the concept of 'rows'

```
all.results$mon  
mean(all.results$mon)
```

all.results				
	"mon"	"tue"	"wed"	"pass"
	1	2	3	4
1	0.8	0.9	0.8	T
2	0.6	0.7	0.5	F
3	0.2	0.3	0.3	F
4	0.8	0.8	0.9	T
5	0.6	1.0	0.9	T

Tibbles are nicer dataframes

```
> head(as.data.frame(data))
```

	Probe	Chromosome	Start	End	Probe	Strand	Feature
1	AL645608.2	1	911435	914948		+	AL645608.2
2	LINC02593	1	916865	921016		-	LINC02593
3	SAMD11	1	923928	944581		+	SAMD11
4	TMEM51-AS1	1	15111815	15153618		-	TMEM51-AS1
5	TMEM51	1	15152532	15220478		+	TMEM51
6	FHAD1	1	15247272	15400283		+	FHAD1

	Description
1	novel transcript
2	long intergenic non-protein coding RNA 2593 [Source:HGNC Symbol;Acc:HGNC:53933]
3	sterile alpha motif domain containing 11 [Source:HGNC Symbol;Acc:HGNC:28706]
4	TMEM51 antisense RNA 1 [Source:HGNC Symbol;Acc:HGNC:26301]
5	transmembrane protein 51 [Source:HGNC Symbol;Acc:HGNC:25488]
6	forkhead associated phosphopeptide binding domain 1 [Source:HGNC Symbol;Acc:HGNC:25489]

Tibbles are nicer dataframes

```
> head(as_tibble(data)) ✓✓  
# A tibble: 6 x 12  
  Probe Chromosome Start End `Probe Strand` Feature ID Description  
  <chr>      <dbl> <dbl> <dbl> <chr>      <chr> <chr> <chr>  
1 AL64~      1 9.11e5 9.15e5 + AL6456~ ENSG~ novel tran~  
2 LINC~      1 9.17e5 9.21e5 - LINC02~ ENSG~ long inter~  
3 SAMD~      1 9.24e5 9.45e5 + SAMD11 ENSG~ sterile al~  
4 TMEM~      1 1.51e7 1.52e7 - TMEM51~ ENSG~ TMEM51 ant~  
5 TMEM~      1 1.52e7 1.52e7 + TMEM51 ENSG~ transmembr~  
6 FHAD1      1 1.52e7 1.54e7 + FHAD1 ENSG~ forkhead a~  
# ... with 4 more variables: `Feature Strand` <chr>, Type <chr>, `Feature  
# Orientation` <chr>, Distance <dbl>
```

Merge Data with R Dplyr

dplyr provides a nice and convenient way to combine datasets. We may have many sources of input data, and at some point, we need to combine them. A join with dplyr adds variables to the right of the original dataset.

Function	Objective	Arguments	Multiple keys
<code>left_join()</code>	Merge two datasets. Keep all observations from the origin table	<code>data, origin, destination, by = "ID"</code>	<code>origin, destination, by = c("ID", "ID2")</code>
<code>right_join()</code>	Merge two datasets. Keep all observations from the destination table	<code>data, origin, destination, by = "ID"</code>	<code>origin, destination, by = c("ID", "ID2")</code>
<code>inner_join()</code>	Merge two datasets. Excludes all unmatched rows	<code>data, origin, destination, by = "ID"</code>	<code>origin, destination, by = c("ID", "ID2")</code>
<code>full_join()</code>	Merge two datasets. Keeps all observations	<code>data, origin, destination, by = "ID"</code>	<code>origin, destination, by = c("ID", "ID2")</code>

Dplyr Joins

We will study all the joins types via an easy example.

First of all, we build two datasets. The table 1 contains two variables, ID, and y, whereas Table 2 gathers ID and z. In each situation, we need to have a **key-pair** variable. In our case, ID is our **key** variable. The function will look for identical values in both tables and bind the returning values to the right of table 1.

Table 1

ID	y
A	5
B	5
C	8
D	0
F	9

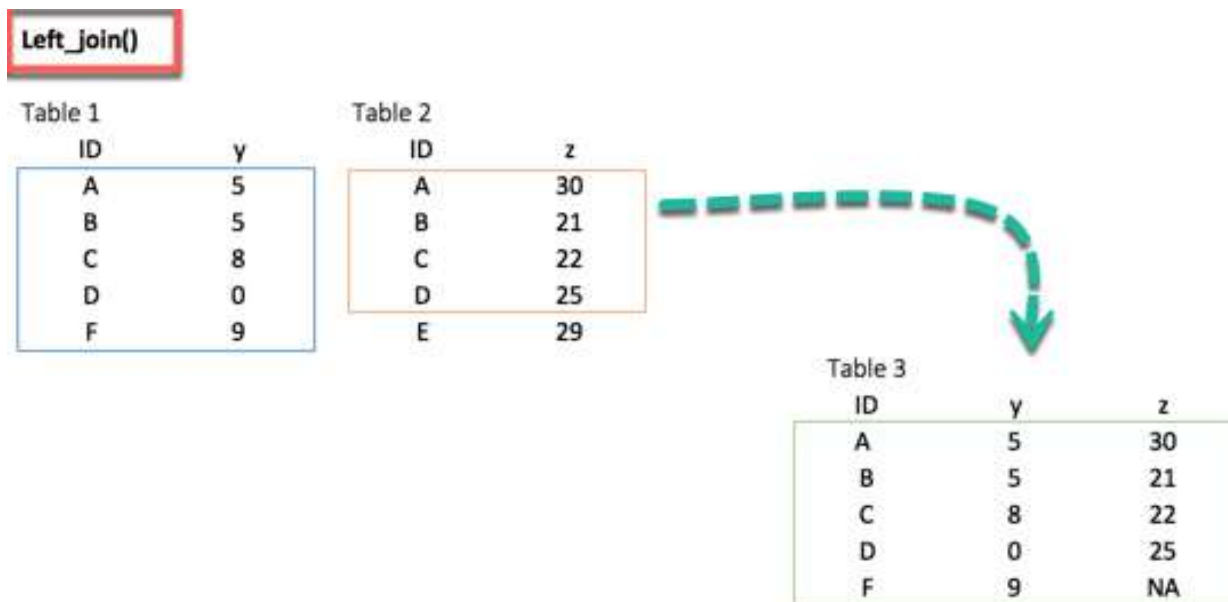
Table 2

ID	z
A	30
B	21
C	22
D	25
E	29

```
library(dplyr)
df_primary <- tribble( ~ID, ~y, "A", 5, "B", 5,
  "C", 8, "D", 0, "F", 9)
df_secondary <- tribble( ~ID, ~z, "A", 30,
  "B", 21, "C", 22, "D", 25, "E", 29)
```

Dplyr left_join()

The most common way to merge two datasets is to use the `left_join()` function. We can see from the picture below that the key-pair matches perfectly the rows A, B, C and D from both datasets. However, E and F are left over. How do we treat these two observations? With the `left_join()`, we will keep all the variables in the original table and don't consider the variables that do not have a key-paired in the destination table. In our example, the variable E does not exist in table 1. Therefore, the row will be dropped. The variable F comes from the origin table; it will be kept after the `left_join()` and return NA in the column z. The figure below reproduces what will happen with a `left_join()`.



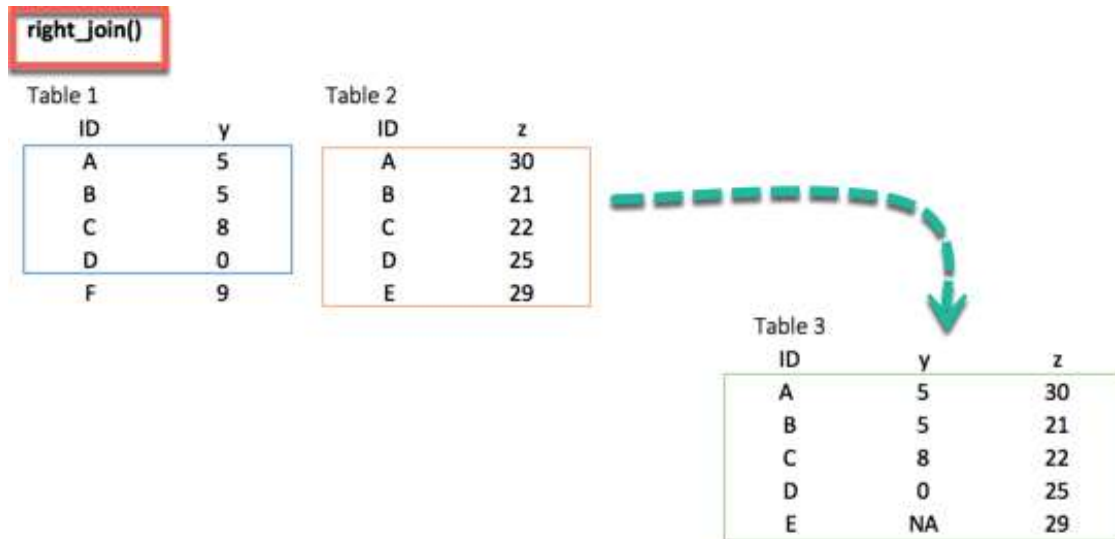
```
left_join(df_primary, df_secondary, by='ID')
```

Output:

```
## # A tibble: 5 x 3
## ID y.x y.y
## <chr> <dbl> <dbl>
## 1 A 5 30
## 2 B 5 21
## 3 C 8 22
## 4 D 0 25
## 5 F 9 NA
```

Dplyr right_join()

The `right_join()` function works exactly like `left_join()`. The only difference is the row dropped. The value E, available in the destination data frame, exists in the new table and takes the value NA for the column y.



```
right_join(df_primary, df_secondary, by = 'ID')
```

Output:

```
##  
# A tibble: 5 x 3  
## ID y.x y.y  
## <chr> <dbl> <dbl>  
## 1 A 5 30  
## 2 B 5 21  
## 3 C 8 22  
## 4 D 0 25  
## 5 E NA 29
```

Dplyr inner_join()

When we are 100% sure that the two datasets won't match, we can consider to return **only** rows existing in **both** dataset. This is possible when we need a clean dataset or when we don't want to impute missing values with the mean or median.

The inner_join() comes to help. This function excludes the unmatched rows.

inner_join()

Table 1

ID	y
A	5
B	5
C	8
D	0
F	9

Table 2

ID	z
A	30
B	21
C	22
D	25
E	29



Table 3

ID	y	z
A	5	30
B	5	21
C	8	22
D	0	25

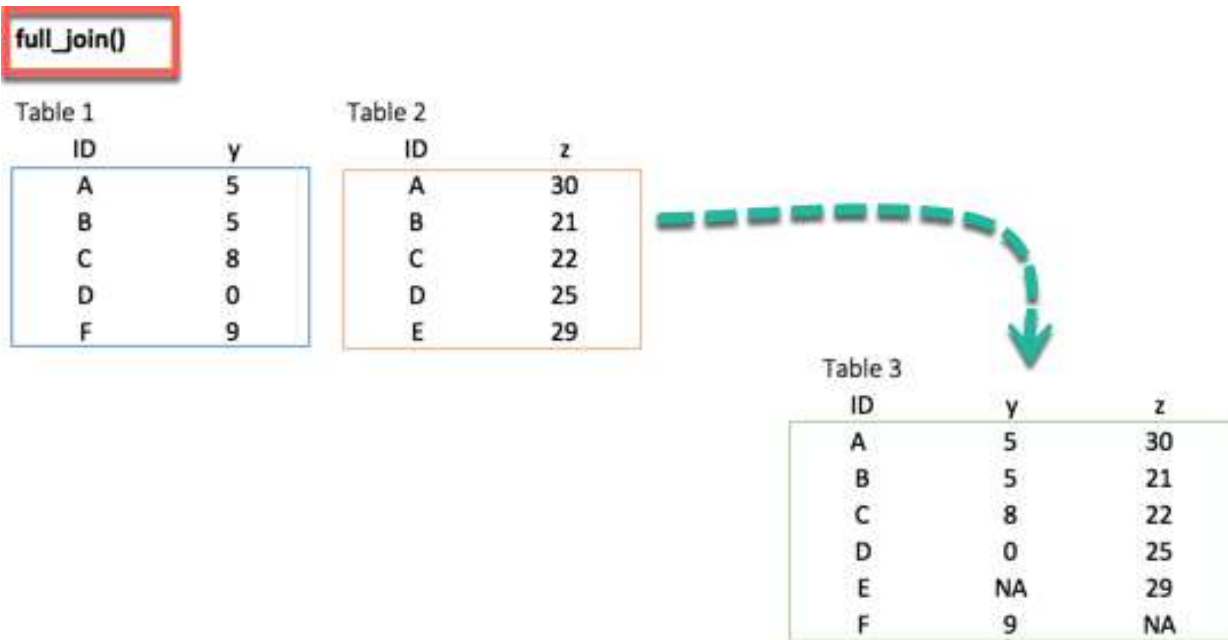
```
inner_join(df_primary, df_secondary, by ='ID')
```

Output:

```
##  
# A tibble: 4 x 3  
## ID y.x y.y  
## <chr> <dbl> <dbl>  
## 1 A 5 30  
## 2 B 5 21  
## 3 C 8 22  
## 4 D 0 25
```

Dplyr full_join()

Finally, the full_join() function keeps all observations and replace missing values with NA.



```
full_join(df_primary, df_secondary, by = 'ID')
```

Output:

```
##  
# A tibble: 6 x 3  
## ID y.x y.y  
## <chr> <dbl> <dbl>  
## 1 A 5 30  
## 2 B 5 21  
## 3 C 8 22  
## 4 D 0 25  
## 5 F 9 NA  
## 6 E NA 29
```

Multiple Key pairs

Last but not least, we can have multiple keys in our dataset. Consider the following dataset where we have years or a list of products bought by the customer.

If we try to merge both tables, R throws an error. To remedy the situation, we can pass two key-pairs variables. That is, ID and year which appear in both datasets. We can use the following code to merge table1 and table 2

Duplicate keys

ID	year	items
A	2015	3
A	2016	7
A	2017	6
B	2015	4
B	2016	8
B	2017	7
C	2015	4
C	2016	6
C	2017	6

ID	year	price
A	2015	9
A	2016	8
A	2017	12
B	2015	13
B	2016	14
B	2017	6
C	2015	15
C	2016	15
C	2017	13

ID	year	items	price
A	2015	3	9
A	2016	7	8
A	2017	6	12
B	2015	4	13
B	2016	8	14
B	2017	7	6
C	2015	4	15
C	2016	6	15
C	2017	6	13

```
df_primary <- tribble( ~ID, ~year, ~items,  
  "A", 2015, 3,  
  "A", 2016, 7,  
  "A", 2017, 6,  
  "B", 2015, 4,  
  "B", 2016, 8,  
  "B", 2017, 7,  
  "C", 2015, 4,  
  "C", 2016, 6,  
  "C", 2017, 6)  
df_secondary <- tribble( ~ID, ~year, ~prices,  
  "A", 2015, 9,  
  "A", 2016, 8,  
  "A", 2017, 12,  
  "B", 2015, 13,  
  "B", 2016, 14,  
  "B", 2017, 6,  
  "C", 2015, 15,  
  "C", 2016, 15,  
  "C", 2017, 13)  
left_join(df_primary, df_secondary, by = c('ID', 'year'))
```


'Tidy' Data Format

- Tibbles give you a 2D data structure where each column must be of a fixed data type
- Often data can be put into this sort of structure in more than one way
- Is there a right / wrong way to structure your data?
- Tidyverse has an opinion!



Journal of Statistical Software
August 2014, Volume 59, Issue 10. <http://www.jstatsoft.org/>

Tidy Data

Hadley Wickham
RStudio

Wide Format

Gene	WT_1	WT_2	WT_3	KO_1	KO_2	KO_3
ABC1	8.86	4.18	8.90	4.00	14.52	13.39
DEF1	29.60	41.22	36.15	11.18	16.68	1.64

- Compact
- Easy to read
- Shows linkage for genes
- No explicit genotype or replicate
- Values spread out over multiple rows and columns
- Not extensible to more metadata

Long Format



Gene	Genotype	Replicate	Value
ABC1	WT	1	8.86
ABC1	WT	2	4.18
ABC1	WT	3	8.90
ABC1	KO	1	4.00
ABC1	KO	2	14.52
ABC1	KO	3	13.39
DEF1	WT	1	29.60
DEF1	WT	2	41.22
DEF1	WT	3	36.15
DEF1	KO	1	11.18
DEF1	KO	2	16.68
DEF1	KO	3	1.64

- More verbose (repeated values)
- Explicit genotype and replicate
- All values in a single column
- Extensible to more metadata

Data Cleaning Functions

Following are the four important functions to tidy (clean) the data:

Function	Objective	Arguments
<code>gather()</code>	Transform the data from wide to long	<code>(data, key, value, na.rm = FALSE)</code>
<code>spread()</code>	Transform the data from long to wide	<code>(data, key, value)</code>
<code>separate()</code>	Split one variables into two	<code>(data, col, into, sep= “”, remove = TRUE)</code>
<code>unit()</code>	Unit two variables into one	<code>(data, col, conc ,sep= “”, remove = TRUE)</code>

We use the tidyr library. This library belongs to the collection of the library to manipulate, clean and visualize the data. If we install R with anaconda, the library is already installed. We can find the library here, <https://anaconda.org/r/r-tidyr>.

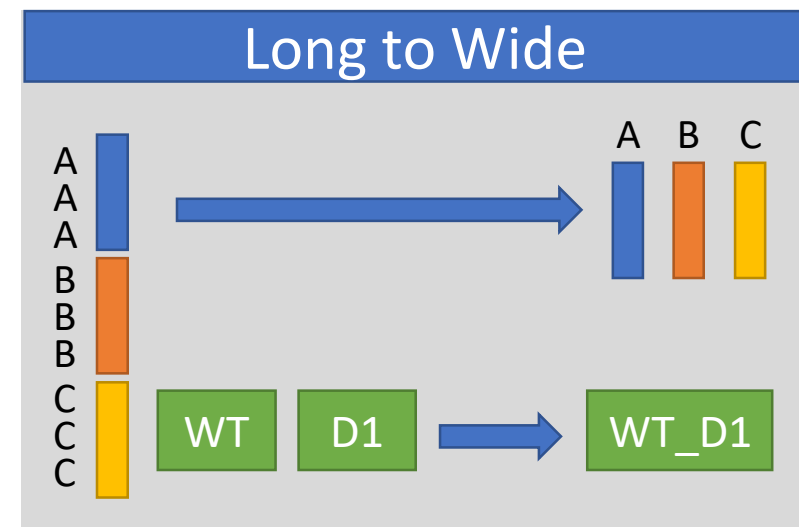
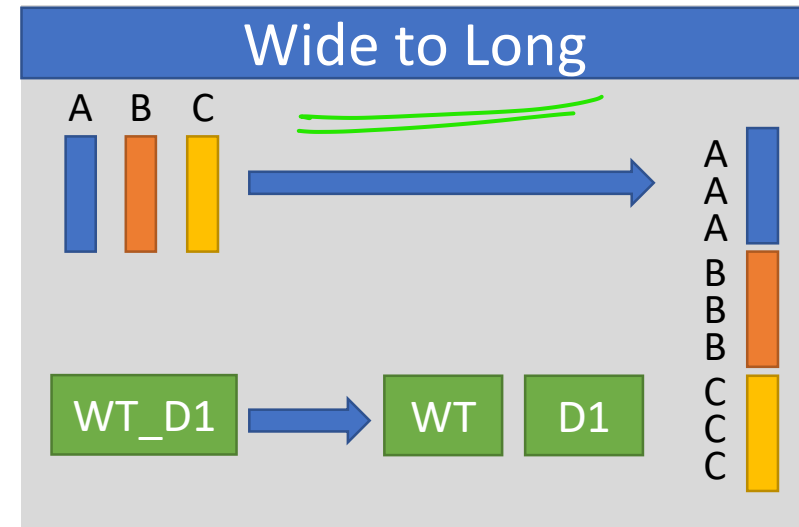
If not installed already, enter the following command to install tidyr:

```
install tidyr : install.packages("tidyr")
```



Tidying operations

- **pivot_longer** ✓
 - Takes multiple columns of the same type and puts them into a pair of key-value columns
- **separate**
 - Splits a delimited column into multiple columns
- **pivot_wider**
 - Takes a key-value column pair and spreads them out to multiple columns of the same type
- **unite**
 - Combines multiple columns into one





Converting to "Tidy" format

```
# A tibble: 3 x 8
  Gene      Chr  Start      End  WT_1  WT_2  KO_1  KO_2
  <chr> <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
1 Gnai3     2  163898  167465  9.39  10.9  33.5  81.9
2 Pbsn      5  4888573  4891351  91.7  59.6  45.3  82.3
3 Cdc45     7 1250084 1262669  69.2  36.1  54.4  38.1
```

- Put all measures into a single column
- Add a 'genotype' and 'replicate' column
- Duplicate the gene information as required
 - Or separate it into a different table



Converting to "Tidy" format

```
# A tibble: 3 x 8
```

	Gene	Chr	Start	End	WT_1	WT_2	KO_1	KO_2
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Gnai3	2	163898	167465	9.39	10.9	33.5	81.9
2	Pbsn	5	4888573	4891351	91.7	59.6	45.3	82.3
3	Cdc45	7	1250084	1262669	69.2	36.1	54.4	38.1

```
non.normalised %>%
```

```
pivot_longer(cols=WT_1:KO_2, names_to="sample", values_to="value") %>%  
separate(sample,into=c("genotype","replicate"),convert = TRUE,sep="_")
```



convert=TRUE makes separate re-detect the type of the column, so replicate becomes a numeric value



Converting to "Tidy" format

```
# A tibble: 12 x 7
```

	Gene	Chr	Start	End	genotype	replicate	value
	<chr>	<dbl>	<dbl>	<dbl>	<chr>	<int>	<dbl>
1	Gnai3	2	163898	167465	WT	1	9.39
2	Pbsn	5	4888573	4891351	WT	1	91.7
3	Cdc45	7	1250084	1262669	WT	1	69.2
4	Gnai3	2	163898	167465	WT	2	10.9
5	Pbsn	5	4888573	4891351	WT	2	59.6
6	Cdc45	7	1250084	1262669	WT	2	36.1
7	Gnai3	2	163898	167465	KO	1	33.5
8	Pbsn	5	4888573	4891351	KO	1	45.3
9	Cdc45	7	1250084	1262669	KO	1	54.4
10	Gnai3	2	163898	167465	KO	2	81.9
11	Pbsn	5	4888573	4891351	KO	2	82.3
12	Cdc45	7	1250084	1262669	KO	2	38.1



Pivoting Examples

```
> pivot.data
# A tibble: 4 x 3
  gene      WT      KO
  <chr> <dbl> <dbl>
1 ABC1  18608  7831
2 DEF1  31988  55502
3 GHI1   7647  93299
4 JKL1  96002  47945
```

```
pivot.data %>%
  pivot_longer(
    cols=WT:KO,
    names_to = "Condition",
    values_to = "Count"
  ) -> pivot.long
```

```
# A tibble: 8 x 3
  gene Condition Count
  <chr> <chr>      <dbl>
1 ABC1  WT        18608
2 ABC1  KO         7831
3 DEF1  WT        31988
4 DEF1  KO        55502
5 GHI1  WT         7647
6 GHI1  KO        93299
7 JKL1  WT        96002
8 JKL1  KO        47945
```



Pivoting Examples

```
> pivot.long
# A tibble: 8 x 3
  gene Condition Count
  <chr> <chr>      <dbl>
1 ABC1  WT         14.2
2 ABC1  KO         12.9
3 DEF1  WT         15.0
4 DEF1  KO         15.8
5 GHI1  WT         12.9
6 GHI1  KO         16.5
7 JKL1  WT         16.6
8 JKL1  KO         15.5
```

```
pivot.long %>%
  pivot_wider(
    names_from = Condition,
    values_from = Count
  )
```

```
# A tibble: 4 x 3
  gene      WT      KO
  <chr> <dbl> <dbl>
1 ABC1  14.2  12.9
2 DEF1  15.0  15.8
3 GHI1  12.9  16.5
4 JKL1  16.6  15.5
```

Missing values

Missing values in data science arise when an observation is missing in a column of a data frame or contains a character value instead of numeric value. Missing values must be dropped or replaced in order to draw correct conclusion from the data.

Exclude Missing Values (NA)

The `na.omit()` method from the `dplyr` library is a simple way to exclude missing observation. Dropping all the NA from the data is easy but it does not mean it is the most elegant solution. During analysis, it is wise to use variety of methods to deal with missing values

To tackle the problem of missing observations, we will use the `titanic` dataset. In this dataset, we have access to the information of the passengers on board during the tragedy. This dataset has many NA that need to be taken care of. We will upload the csv file from the internet and then check which columns have NA. To return the columns with missing data, we can use the following code:

Let's upload the data and verify the missing data.

```
df_titanic <- read.csv(PATH, sep = ",")  
# Return the column names containing missing observations  
list_na <- colnames(df_titanic)[ apply(df_titanic, 2, anyNA) ]  
list_na
```

Output:

```
## [1] "age" "fare"
```

1 = Row
2 = Column

na.omit()

The columns age and fare have missing values.
We can drop them with the na.omit().

```
library(dplyr)
# Exclude the missing observations
df_titanic_drop <- df_titanic %>% na.omit()
dim(df_titanic_drop)
```

Output:

```
## [1] 1045 13
```


The new dataset contains 1045 rows compared to 1309 with the original dataset.

57	male	36.0000	1	2	113760	120.0000	B9
58	female	36.0000	1	2	113760	120.0000	B9
59	male	49.0000	0	0	19924	26.0000	
60	female	NA	0	0	17770	27.7208	
61	male	36.0000	1	0	19877	78.8500	
62	female	76.0000	1	0	19877	78.8500	
63	male	46.0000	1	0	W.E.P.	5734	61.1750
64	female	47.0000	1	0	W.E.P.	5734	61.1750
65	male	27.0000	1	0	113806	53.1000	
66	female	33.0000	1	0	113806	53.1000	
67	female	36.0000	0	0	PC	17608	262.3750
68	female	30.0000	0	39	male	41.0000	113034
69	male	45.0000	0	40	male	48.0000	17591
70	female	NA	0	42	female	44.0000	17610
71	male	NA	0	43	female	59.0000	11769
72	male	27.0000	0	44	female	60.0000	11813
73	female	26.0000	1	45	female	41.0000	16966
74	female	22.0000	0	46	male	45.0000	113050
75	male	NA	0	48	male	42.0000	17476
76	male	47.0000	0	49	female	53.0000	17606
				50	male	36.0000	17755
				51	female	58.0000	17755
				52	male	33.0000	695
				53	male	28.0000	113059
				54	male	17.0000	113059
				55	male	11.0000	113760
				56	female	14.0000	113760
				57	male	36.0000	113760
				58	female	36.0000	113760
				59	male	49.0000	19924
				61	male	36.0000	19877

Impute Missing data with the Mean and Median

A big data set could have lots of missing values and the above method could be cumbersome. We can execute all the above steps above in one line of code using `sapply()` method. Though we would not know the values of mean and median. `sapply` does not create a data frame, so we can wrap the `sapply()` function within `data.frame()` to create a data frame object.

```
# Quick code to replace missing values with the mean  
  
df_titanic_impute_mean <- data.frame( sapply( df_titanic, function(x)  
    ifelse(is.na(x), mean(x, na.rm = TRUE), x)))
```



Exercise 1 - Data normalization

Data normalization

1. Create a function that normalizes a vector:
 $x_{\text{new}} = (x - \text{mean}(x)) / \text{sd}(x)$
2. Use this function on the iris dataset so that each column is normalized
3. You can also make the function more general
 $x_{\text{new}} = (x - a) / b$
4. use it to preprocess the data with « min-max » normalization ($x - \text{min} / (\text{max} - \text{min})$)

Exercise 4: mean and standard deviation over the columns

1. Compute the mean of all columns of iris dataset
2. Compute their standard deviation

Exercise 5

Exercise 1

Select the first three columns of the iris dataset using their column names. **HINT:** Use `select()`.

Exercise 2

Select all the columns of the iris dataset except “Petal Width”. **HINT:** Use “-”.

Exercise 3

Select all columns of the iris dataset that start with the character string “P”.

Exercise 4

Filter the rows of the iris dataset for `Sepal.Length >= 4.6` and `Petal.Width >= 0.5`.

Exercise 5

Pipe the iris data frame to the function that will select two columns (`Sepal.Width` and `Sepal.Length`). **HINT:** Use pipe operator.

Exercise 6

Arrange rows by a particular column, such as the `Sepal.Width`. **HINT:** Use `arrange()`.

Exercise 7

Select three columns from iris, arrange the rows by `Sepal.Length`, then arrange the rows by `Sepal.Width`.

Exercise 8

Create a new column called `proportion`, which is the ratio of `Sepal.Length` to `Sepal.Width`. **HINT:** Use `mutate()`.

Exercise 9

Compute the average number of `Sepal.Length`, apply the `mean()` function to the column `Sepal.Length`, and call the summary value “`avg_slength`”. **HINT:** Use `summarize()`.