R programming
Module 1: Introduction to R Programming

Learning objectives

1. Master the use of the R interactive environment.
2. Expand R by installing R packages.
3. Understand the different arithmetic operations, variables and data types in R.
4. Use R for mathematical operations.
5. Use of vectorized calculations, matrix and factors.
6. Understand the use of `data.frames` and `lists`.
7. You will learn about operators, conditional statements.
8. Understand loops and functions to power your own R scripts.
9. You will learn more efficiently and readable using the apply functions.
10. This R module will allow you to take the next step in advancing your overall knowledge and capabilities while programming in R.

History

1. R is an implementation of the S programming language. S was created by John Chambers in 1976, while at Bell Labs.

2. There are some important differences between them, but much of the code written for S runs unaltered in R.

3. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and currently developed by the R Development Core Team.

4. R is named partly after the first names of the first two R authors and partly as a play on the name of S. The project was conceived in 1992, with an initial version released in 1995 and a stable beta version in 2000.

Introduction

1. R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing.

2. The R language is widely used among statisticians and data miners for developing statistical software and data analysis.

3. R alone has been used for projects with banks, political campaigns(Polls), tech startups and aid organizations, hospitals and real estate developers.

4. Data mining surveys and studies of scholarly literature databases show substantial increases in popularity in recent years. As of January 2019, R ranks 12th in the TIOBE index (a measure of the popularity of programming languages).

The following is a list of top brands or large organizations using R.

1. Facebook – For behavior analysis related to status updates and profile pictures.
2. Google – For advertising effectiveness and economic forecasting.
3. Twitter – For data visualization and semantic clustering
4. Microsoft – Acquired Revolution of R company and use it for a variety of purposes.
5. Uber – For statistical analysis

Installing R and RStudio

To use R, you first need to install the R program on your computer. Unlike with languages such as C and C++, R must be installed in order to run.

What is RStudio?
RStudio is a free and open-source integrated development environment (IDE) for R, RStudio was founded by JJ Allaire, creator of the programming language ColdFusion. Hadley Wickham is the Chief Scientist at RStudio.

RStudio is available in two editions: RStudio Desktop, where the program is run locally as a regular desktop application; and RStudio Server, which allows accessing RStudio using a web browser while it is running on a remote Linux server. Prepackaged distributions of RStudio Desktop are available for Windows, macOS, and Linux. RStudio is available in open source and commercial editions and runs on the desktop (Windows, macOS, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro (Debian, Ubuntu, Red Hat Linux, CentOS, openSUSE and SLES).

RStudio is partly written in the C++ programming language and uses the Qt framework for its graphical user interface. The bigger percentage of the code is written in Java. JavaScript is also amongst the languages used. Work on RStudio started around December 2010, and the first public beta version (v0.92) was officially announced in February 2011. Version 1.0 was released on 1 November 2016. Version 1.1 was released on 9 October 2017

Downloading R and RStudio

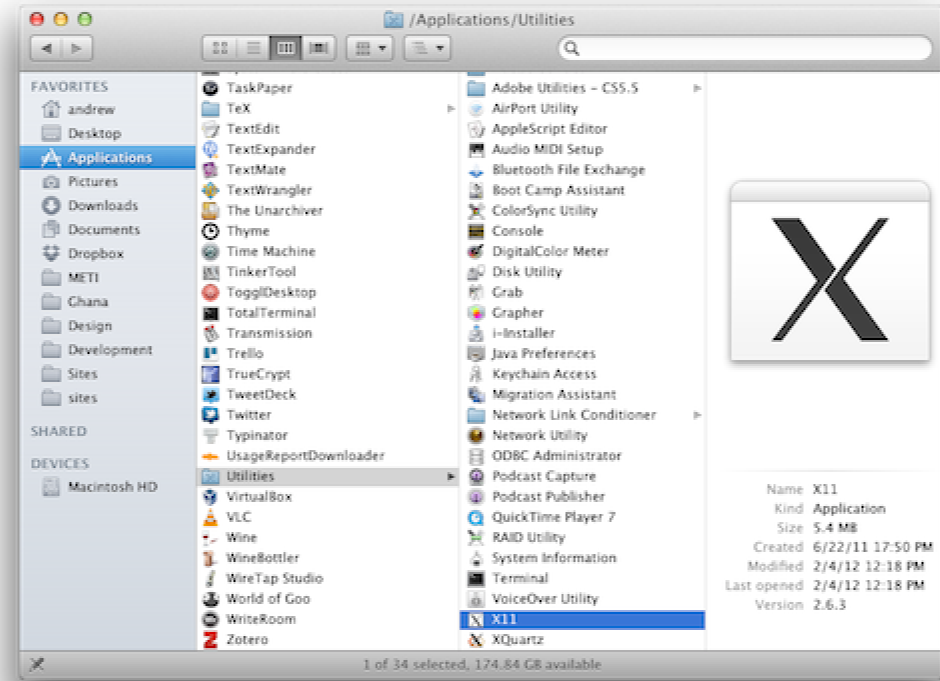
The program is easily obtainable from the Comprehensive R Archive Network (CRAN), the maintainer of R. There are the links to download R for Windows, Mac OS X, and Linux.

- **To install R on your Windows computer**

    1. Go to http://cran.r-project.org/
    2. Under "Download and Install R", click on the "Windows" link.
    3. Under "Subdirectories", click on the "base" link.
    4. On the next page, you should see a link saying something like "Download R 3.7.2 for Windows" (or R X.X.X, where X.X.X gives the version of R). Click on this link.
    5. To Install RStudio, go to www.rstudio.com and click on the "Download RStudio" button.
    6. Click on "Download" RStudio Desktop.
    7. Click on the version recommended for your system, or the latest Windows version, and save the executable file. Run the .exe file and follow the installation instructions.

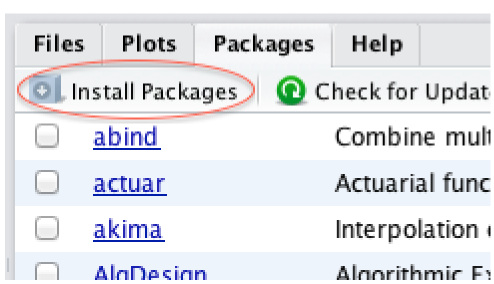
- **To install R on Mac OS X**

    1. Download R from http://cran.r-project.org/ (click on "Download R for Mac OS X" > "R-3.7.2pkg (latest version)")
    2. Install R.
    3. Download RStudio from http://rstudio.org/download/desktop.
    4. Install RStudio by dragging the application icon to your Applications folder.
    5. Download Tcl/Tk from http://cran.r-project.org/bin/macosx/tools/ (click on tcltk-8.x.x-x11.dmg; OS X needs this to run R Commander.)
    6. Install Tcl/Tk.
    7. Go to your Applications folder and find a folder named Utilities. Verify that you have a program named "X11" there. If not, go to http://xquartz.macosforge.org/ and download and install the latest version of XQuartz.

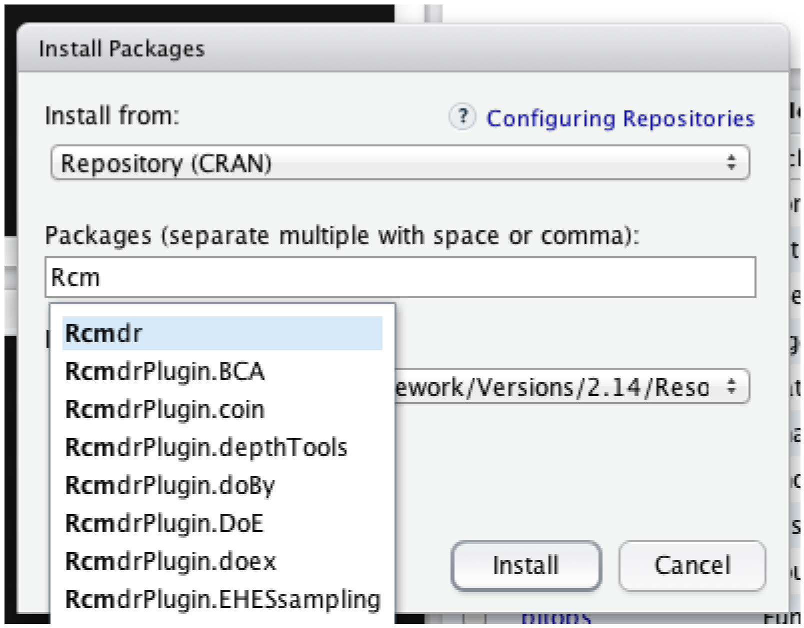


8. Open RStudio.

9. Go to the “Packages” tab and click on “Install Packages”. The first time you’ll do this you’ll be prompted to choose a CRAN mirror. R will download all necessary files from the server you select here. Choose the location closest to you (probably “USA CA 1” or “USA CA 2”, which are housed at UC Berkeley and UCLA, respectively).



10. Start typing “`Rcmdr`” until you see it appear in a list. Select the first option (or finish typing Rcmdr), ensure that “Install dependencies” is checked, and click “Install”.



11. Wait while all the parts of the R Commander package are installed.

Open R Commander in Windows and OS X.

Once you've installed R Commander, you won't have to go through all those steps again! Running R Commander from this point on is simple—follow the instructions below.

If you decide to stop using R Commander and just stick with R, all you ever need to do is open RStudio—even simpler!

1. Open R Studio
2. In the console, type `windows()` if using Windows, `quartz()` if using Mac OS X. (This tells R Commander to output all graphs to a new window). If you don't do this, R Commander graphs will be output to the graphics window in RStudio.
3. Go to the "Packages" tab, scroll down to "Rcmdr," and check the box to load the plugin. (Alternatively, type library(Rcmdr) at the console.)

- **To Install R on Linux Ubuntu 16.04**

1. Add R repository

First, we've got to add a line to our */etc/apt/sources.list* file. This can be accomplished with the following. Note the "xenial" in the line, indicating Ubuntu 16.04. If you have a different version, just change that.

```
sudo echo "deb http://cran.rstudio.com/bin/linux/ubuntu xenial/" | sudo tee -a /etc/apt/sources.list
```

2. Add R to Ubuntu Keyring

First : `gpg --keyserver keyserver.ubuntu.com --recv-key E084DAB9`
Then : `gpg -a --export E084DAB9 | sudo apt-key add -`

3. Install R-Base
Most Linux users should be familiar with the old…

```
sudo apt-get update
sudo apt-get install r-base r-base-dev
```

4. Installing R-Studio
From here you can download your files and install the IDE through Ubuntu Software Center or Synaptic Package Manager, or since you've already got the terminal open, you could just:

```
sudo apt-get install gdebi-core
wget https://download1.rstudio.org/rstudio-0.99.896-amd64.deb
sudo gdebi -n rstudio-0.99.896-amd64.deb
rm rstudio-0.99.896-amd64.deb
```

> *At this point, R is fully usable and comes with a crude GUI. However, it is best to install RStudio and use its interface. The process involves downloading and launching an installer, just as with any other program.*

Basics of R

R is a powerful tool for all manner of calculations, data manipulation and scientific computations. Like most languages, R has its share of mathematical capability, variables, functions and data type.

Arithmetic with R

In its most basic form where R can be used as a simple calculator. These are the following arithmetic operators:

1. Addition: +
2. Subtraction: -
3. Multiplication: *
4. Division: /
5. Exponentiation: ^ : The ^ operator raises the number to its left to the power of the number to its right: for example, 3^2 is 9.
6. Modulo: %% : The modulo returns the remainder of the division of the number to the left by the number on its right, for example, 5 modulo 3 or 5 %% 3 is 2.

Example

```
# An addition
```

```
> 5 + 5
[1] 10

# A subtraction
> 5 - 5
[1] 0

# A multiplication
> 3 * 5
[1] 15

# A division
> (5 + 5) / 2
[1] 5
```

Variables

Variables are an integral part of any programming language and R offers a great deal of flexibility. R does not require variable types to be declared. It also holds any R object such as a function or the result of analysis or a plot. A single variable can at one point hold a number, then later hold a character.

Variable assignment

There are a number of ways to assign a value to a variable.

```
> Var <- 2
> Var
[1] 2

> X = 5
> X
[1] 5

> 4 <- Y
> Y
[1] 4

> a <- b <- 7
> a
[1] 7
> b
[1] 7

> assign(“j”,4)
> j
[1] 4
```

Variable names may contain any combination of alphanumeric characters along with period (.) and underscores (_). They can’t start with a number or an underscore.

Removing variables

For various reasons a variable may need to be removed. This is easily done using remove or its shortcut **rm()**.

```
> j
[1] 4

> rm(j)

> j
Error: object 'j' not found
```

Data type

There are numerous data types in R that store various kinds of data. The four main types of data most likely to be used are numeric, character (string), logical (TRUE/FALSE).

The type of data contained in a variable is checked with the class function.

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

Numeric Data

R excels at running numbers, so numeric data is the most common type in R. The most commonly used numeric data is numeric. This is similar to a float or double in other languages. It handles integers and decimals, both positive and negative, and, of course, zero. A numeric value stored in a variable is automatically assumed to be numeric. Testing whether a variable is numeric is done with the function `is.numeric`.

```
> is.numeric(x)
[1] TRUE
```

As the name implies this is for whole numbers only, no decimals. To set an integer to a variable it is necessary to append the value with an L. As with checking for a numeric, the `is.integer` function is used.

```
> i <- 5L
> i
[1] 5

> is.integer(i)
[1] TRUE
```

R nicely promotes integers to numeric when needed. This is obvious when multiplying an integer by a numeric, but importantly it works when dividing an integer by another integer, resulting in a decimal number.

```
> class(4L)
[1] "integer"
```

```
> class(2.8)
[1] "numeric"

> 4L * 2.8
[1] 11.2

> class(4L * 2.8)
[1] "numeric"

> class(5L)
[1] "integer"

> class(2L)
[1] "integer"
```

Character Data
Even though it is not explicitly mathematical, the character (string) data type is very common in statistical analysis and must be handled with care. R has two primary ways of handling character data: character and factor.

```
> x <- "data"
> x
[1] "data"
```

Characters are case sensitive, so `“Data”` is different from `“data”` or `“DATA”`. To find the length of a character (or numeric) use the nchar function.

```
> nchar(x)
[1] 4

> nchar("hello")
[1] 5

> nchar(3)
[1] 1

> nchar(452)
[1] 3
```

Logical Data
Logicals are a way of representing data that can be either TRUE or FALSE. Numerically, TRUE is the same as 1 and FALSE is the same as 0. So TRUE * 5 equals 5 while FALSE * 5 equals 0.

```
> TRUE * 5
[1] 5

> FALSE * 5
```

```
[1] 0
```

Similar to other types, logicals have their own test, using the `is.logical` function.

```
> k <- TRUE
> class(k)
[1] "logical"

> is.logical(k)
[1] TRUE

> # does 2 equal 3?
> 2 == 3
[1] FALSE

> # does 2 not equal three?
> 2 != 3
[1] TRUE

> # is two less than three?
> 2 < 3
[1] TRUE
```

Vectors

Vector is a basic data structure in R. A vector is a collection of elements, all of the same type. The data types can be logical, integer, character. A vector cannot be mixed type. R is a vectorized language. That means operations are applied to each element of the vector automatically, without the need to loop through the vector. Vector do not have a dimension, it means no column or row vector. A vectors type can be checked with the `typeof()` function. We can create, name, select elements and compare different vectors.

Create a vector

Vectors are one-dimension arrays that can hold numeric data, character data or logical data. That means vector is simple tool to store data. In R, you create a vector with the **combine function** `c()`.

```
> num_vector <- c(1,2,3,4,5)
> char_ vector <- c(“A”,”B”,”C”)
> logical_vector <- c(TRUE,FALSE,TRUE,FALSE)
```

Atomic vectors are always flat,even if you nest `c()`'s.

```
 > c(1, c(2, c(3,4)))
[1]  1 2 3 4
```

Same as,

```
 > c(1,2,3,4)
[1] 1 2 3 4
```

Naming a vector

It is possible to give names to a vector either during creation or after the fact. Using **`names()`** function.

```
> c(One="x", Two="y", Last="z")  #provide a name for each element of an array using a name-value

One       Two      Last
"x "      "y"      "z"

# create a vector
> w <- 1:3

# name the elements
> names(w) <- c("a","b","c")

> w
  a    b    c
  1    2    3
```

Vector selection

Accessing individual elements of a vector is done using square brackets ([ ]). The first elements in a vector has index `x[1]`, not `0` as in many other programming languages. The first two elements by `x[1:2]` and non consecutive elements by `x[c(1,4)]`.

```
> x <- c(1,2,3,4,5,6,7,8,9,10)

> x[1]
[1] 1

> x[1:2]
[1]  1  2

> x[c(1,4)]
[1]  1  4
```

Vector operation

Now that we have a vector of the first ten numbers,we might want to multiply each element by 3. In R this is a simple operation using just the multiplication operator (*).

```
> x <- c (1,2,3,4,5,6,7,8,9,10)

> x
[1]  1  2  3  4  5  6  7  8  9  10

> x * 3
[1]  3  6  9  12  15  18  21  24  27  30
```

No loops are necessary. Addition, Subtraction and Division are just easy.

```
> x ^ 2
[1]  1  4  9  16  25  36  49  64  81  100

> sqrt (x)
[1] 1.000  1.414  1.732  2.000  2.236  2.449  2.646  2.828  3.000
3.162
```

Vector operations can be extended even further. Let's say we have two vectors of equal length. Each of the corresponding elements can be operated on together.

```
> a <- c(1,4,9,16)
> b <- c(1,4,9,16)
> a + b
[1] 2  8  18  32

# check the length of each
> length(x)
[1] 10
```

Vector comparisons

Comparisons also work on vectors. Here the result is a vector of the same length containing TRUE or FALSE for each element.

The comparisons operators known to R are:

1. < for less than
2. > for greater than
3. <= for less than or equal to
4. >= for greater than or equal to
5. == for equal to each other
6. != not equal to each other

```
> x <= 5
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  FALSE  FALSE  FALSE  FALSE
FALSE

> x <-  10 : 1
> y <-  -4 : 5
> any ( x < y )
[1] TRUE

> all ( x < y )
[1] FALSE
```

Matrix

A very common mathematical structure that is essential to statistics is a matrix. In R, a matrix is a collection of elements of the same data type arranged into a fixed number of rows and columns. You can construct a matrix in R with the **matrix()** function. Matrix act similarly to vectors with element by element addition, multiplication, subtraction, division and equality.

```
> matrix ( 1:9, byrow = TRUE, nrow = 3 )
```

In **matrix()** function

1. The first argument is the collection of elements that R will arrange into the rows and columns of the matrix. Here, we use 1:9 which is a shortcut for c(1, 2, 3, 4, 5, 6, 7, 8, 9).
2. The argument byrow indicates that the matrix is filled by the rows. If we want the matrix to be filled by the columns, we just place byrow = FALSE.
3. The third argument `nrow` indicates that the matrix should have three rows.

```
> A < - matrix ( 1 : 10 ,  nrow = 5 )
> A
         [,1]    [,2]
[1,]      1       6
[2,]      2       7
[3,]      3       8
[4,]      4       9
[5,]      5       10

> nrow (A)
[1]  5

> ncol ( A)
[1]  2

> dim (A)
[1]  5  2

> B <- matrix ( 1:10 , nrow = 5)
> B
         [,1]    [,2]
[1,]      1       6
[2,]      2       7
[3,]      3       8
[4,]      4       9
[5,]      5       10

> A + B
         [,1]    [,2]
[1,]      2       12
[2,]      4       14
```

```
[3,]      6       16
[4,]      8       18
[5,]      10      20
```

```
> A * B
         [,1]    [,2]
[1,]      1       36
[2,]      4       49
[3,]      9       64
[4,]      16      81
[5,]      25      100
```

```
> A == B
        [,1]       [,2]
[1,]    TRUE       TRUE
[2,]    TRUE       TRUE
[3,]    TRUE       TRUE
[4,]    TRUE       TRUE
[5,]    TRUE       TRUE
```

Naming a matrix

```
> colnames (A)
NULL

> rownames (A)
NULL

> colnames (A) <-  (“left”, “right”)
> rownames (A) <- ( “1st”, “2nd”, “3rd”, “4th” , “5th”)

>A
         left    right
1st      1       6
2nd      2       7
3rd      3       8
4th      4       9
5th      5       10
```

Factor

Factor is a data structure used for fields that takes only pre-defined, finite number of values (categorical data). For example: a data field such as marital status may contain only values from single, married, separated, divorced, or widowed. In such case, we know the possible values beforehand and these predefined, distinct values are called levels. Following is an example of factor in R.

```
> x
[1] single married married single
Levels: married single
```

Here, we can see that factor x has four elements and two levels. We can check if a variable is a factor or not using class() function.
Similarly, levels of a factor can be checked using the `levels()` function.

```
> class(x)
[1] "factor"

> levels(x)
[1] "married" "single”
```

We can create a factor using the function `factor()`. Levels of a factor are inferred from the data if not provided.

```
> x <- factor(c("single", "married", "married", "single"));
> x
[1] single  married married single
Levels: married single

> x <- factor(c("single", "married", "married", "single"), levels = c("single", "married", "divorced"));

> x
[1] single  married married single
Levels: single married divorced
```

In addition to basic calculations, R can handle numeric, character and time-based data. One of the nicer parts of working with R, although one that requires a different way of thinking about programming, is vectorization. This allows operating on multiple elements in a vector simultaneously, which leads to faster and more mathematical code.

Data Frames

One of the most useful features of R is the data.frame. It is one of the most often cited reasons for R’s ease of use. On the surface a data.frame is just like an Excel spreadsheet in that it has columns and rows. In statistical terms, each column is a variable and each row is an observation.

In terms of how R organizes `data.frames`, each column is actually a vector, each of which has the same length. That is very important because it lets each column hold a different type of data. This also implies that within a column each element must be of the same type, just like with vectors.

There are numerous ways to construct a `data.frame`, the simplest being to use the `data.frame` function.

```
> x <- 10 : 1
> y <- -4 : 5
> q <- c( "Hockey", "Football", "Baseball", "Curling", "Rugby",
"Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")

> theDF <- data.frame (x, y, q)

> theDF
    x       y          q
1  10      -4       Hockey
2   9      -3       Football
3   8      -2       Baseball
4   7      -1       Curling
5   6       0       Rugby
6   5       1       Lacrosse
7   4       2       Basketball
8   3       3       Tennis
9   2       4       Cricket
10  1       5       Soccer
```

This creates a 10x3 data.frame consisting of those three vectors. Notice the names of `theDF` are simply the variables. We could have assigned names during the creation process, which is generally a good idea.

```
> theDF <- data.frame(First = x, Second = y, Sport = q)

> theDF
  First   second     Sport
1  10      -4       Hockey
2   9      -3       Football
3   8      -2       Baseball
4   7      -1        Curling
5   6       0        Rugby
6   5       1       Lacrosse
7   4       2       Basketball
8   3       3       Tennis
9   2       4       Cricket
10  1       5       Soccer
```

data.frames are complex objects with many attributes. The most frequently checked attributes are the number of rows and columns. Of course there are functions to do this for us: `nrow` and `ncol`. And in case both are wanted at the same time there is the dim function.

```
> nrow(theDF)
[1] 10

> ncol(theDF)
```

```
[1] 3

> dim(theDF)
[1] 10 3
```

Checking the column names of a data.frame is as simple as using the names function. This returns a character vector listing the columns. Since it is a vector we can access individual elements of it just like any other vector.

```
> names(theDF)
[1] "First" "Second" "Sport"

> names(theDF)[3]
[1] "Sport"
```

We can also check and assign the row names of a data.frame.

```
> rownames(theDF)
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

> rownames(theDF) <- c("One", "Two", "Three", "Four", "Five", "Six",
"Seven", "Eight", "Nine", "Ten")

> rownames(theDF)
```

Usually a data.frame has far too many rows to print them all to the screen, so thankfully the head function prints out only the first few rows.

```
> head(theDF)
    First   second     Sport
1  10        -4        Hockey
2   9        -3        Football
3   8        -2        Baseball
4   7        -1        Curling
5   6         0        Rugby
6   5         1        Lacrosse

> head(theDF, n = 7)
 First   second        Sport
1  10        -4        Hockey
2   9        -3        Football
3   8        -2        Baseball
4   7        -1        Curling
5   6         0        Rugby
6   5         1        Lacrosse
7   4         2        Basketball

> tail(theDF)
```

```
5   6         0         Rugby
6   5         1         Lacrosse
7   4         2         Basketball
8   3         3         Tennis
9   2         4         Cricket
10  1         5         Soccer
```

Similar to `vectors`, `data.frames` allow us to access individual elements by their position using square brackets, but instead of having one position two are specified. The first is the row number and the second is the column number. So to get the third row from the second column we use the `DF[3,2]`.

To specify more than one row or column use a vector of indices.

```
> theDF[3, 2]
[1] -2

> # row 3, columns 2 through 3
> theDF[3, 2:3]
                second     Sport
     3           -2       Baseball

> # rows 3 and 5, column 2
> # since only one column was selected it was returned as a vector
> # hence the column names will not be printed
> theDF[c(3, 5), 2]
[1]  -2  0
```

Lists

Often a container is needed to hold arbitrary objects of either the same type or varying types. R accomplishes this through lists. They store any number of items of any type. A list can contain all numerics or characters or a mix of the two or data.frames or, recursively, other lists.

Lists are created with the list function where each argument to the function becomes an element of the list.

```
# creates a three element list
> list(1, 2, 3)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

# creates a single element list where the only element is a vector
# that has three elements
```

```
> list(c(1, 2, 3))
[[1]]
[1] 1 2 3

# creates a two element list
# the first element is a three element vector
# the second element is a five element vector
(list3 <- list(c(1, 2, 3), 3:7))
[[1]]
[1] 1 2 3

[[2]]
[1] 3 4 5 6 7

> # two element list
> # first element is a data.frame
> # second element is a 10 element vector
> list(theDF, 1:10)
[[1]]
    x     y        q
1  10    -4      Hockey
2   9    -3      Football
3   8    -2      Baseball
4   7    -1      Curlin
5   6     0      Rugby
6   5     1      Lacrosse
7   4     2      Basketball
8   3     3      Tennis
9   2     4      Cricket
10  1     5      Soccer

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

# three element list
# first is a data.frame
# second is a vector
# third is list3, which holds two vectors
> list5 <- list(theDF, 1:10, list3)
> list5
[[1]]
    x     y        q
1  10    -4      Hockey
2   9    -3      Football
3   8    -2      Baseball
```

```
4   7    -1      Curlin
5   6     0      Rugby
6   5     1      Lacrosse
7   4     2      Basketball
8   3     3      Tennis
9   2     4      Cricket
10  1     5      Soccer

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

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

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

Notice in the previous block of code (where `list3` was created) that enclosing an expression in parentheses displays the results after execution.

Like `data.frames`, `lists` can have names. Each element has a unique name that can be either viewed or assigned using `names`.

```
> names(list5)
> NULL
> names(list5) <- c("data.frame", "vector", "list")
> names(list5)
[1] "data.frame"  "vector"    "list"
list5
$data.frame
  First second   Sport
1  10    -4     Hockey
2   9    -3     Football
3   8    -2     Baseball
4   7    -1     Curlin
5   6     0     Rugby
6   5     1     Lacrosse
7   4     2     Basketball
8   3     3     Tennis
9   2     4     Cricket
10  1     5     Soccer

$vector
[1]  1  2  3  4  5  6  7  8  9 10

$list
```

```
$list[[1]]
[1] 1 2 3

$list[[2]]
[1] 3 4 5 6 7
```

Data come in many types and structures, which can pose a problem for some analysis environments but R handles them with assurance. The most common data structure is the one-dimensional vector, which forms the basis of everything in R. The most powerful structure is the data.frame—something special in R that most other languages do not have—which handles mixed data types in a spreadsheet-like format. Lists are useful for storing collections of items.

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides many types of operators.
We have the following types of operators in R programming −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Miscellaneous Operators

Arithmetic Operators

```
> v <- c (2, 5.5, 6)
> t <- c (8, 3,  4)

> v + t
[1] 10.0  8.5  10.0

> v - t
[1] -6.0  2.5  2.0

> v * t
[1] 16.0  16.5  24.0

> v / t
[1] 0.250000  1.833333  1.500000
```

Give the remainder of the first vector with the second

```
> v %% t
```

It produces the following result −

```
[1] 2.0  2.5  2.0
```

The first vector raised to the exponent of second vector

```
> v ^ t
```

It produces the following result −

```
[1] 256.000  166.375  1296.000
```

Relational Operators

Checks if each element of the first vector is greater than the corresponding element of the second vector.

```
> v <- c (2, 5.5, 6, 9)
> t <- c (8, 2.5, 14, 9)
> v > t
```

It produces the following result –

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

Checks if each element of the first vector is less than the corresponding element of the second vector.

```
> v < t
```

It produces the following result –

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

Checks if each element of the first vector is equal to the corresponding element of the second vector.

```
> v  ==  t
```

It produces the following result –

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

Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.

```
> v <= t
```

It produces the following result –

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

Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.

```
> v >= t
```

It produces the following result –

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

Checks if each element of the first vector is unequal to the corresponding element of the second vector.

```
> v != t
```

It produces the following result –

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

Logical Operators

It is called Element-wise Logical `AND` operator. It combines each element of the first vector with the corresponding element of the second vector and gives an output `TRUE` if both the elements are `TRUE`.

```
> v <- c ( 3, 1, TRUE, 2+3i )
> t <- c ( 4, 1, FALSE, 2+3i )
> v & t
```

It produces the following result –

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

It is called Element-wise Logical `OR` operator. It combines each element of the first vector with the corresponding element of the second vector and gives an output `TRUE` if one of the elements is `TRUE`.

```
> v | t
```

It produces the following result –

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

It is called Logical `NOT` operator. Takes each element of the vector and gives the opposite logical value.

```
> v <- c (3, 0, TRUE, 2+2i)
> !v
```

It produces the following result –

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

The logical operator `&&` and `||` considers only the first element of the vectors and give a vector of a single element as output.
Operator Description `&&` Called Logical `AND` operator.
Takes the first element of both the vectors and gives the `TRUE` only if both are `TRUE`.

```
> v <- c ( 3, 0, TRUE, 2+2i )
> t <- c ( 1, 3, TRUE, 2+3i )
> v && t
```

It produces the following result –

```
[1] TRUE
```

Operator Description `||` Called Logical `OR` operator. Takes first element of both the vectors and gives the `TRUE` if one of them is `TRUE`.

```
> v <- c (0, 0, TRUE, 2+2i)
> t <- c (0, 3, TRUE, 2+3i)
> v || t
```

It produces the following result –

```
[1] FALSE
```

Miscellaneous Operators

Colon operator (:) It creates the series of numbers in sequence for a vector.

```
> v <- 2:8
> v
```

It produces the following result –

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

`%in%` This operator is used to identify if an element belongs to a vector.

```
> v1 <- 8
> v2 <- 12
> t <- 1:10
> v1 %in% t
> v2 %in% t
```

It produces the following result –

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

`%*%` This operator is used to multiply a matrix with its transpose.

```
> M = matrix( c(2,6,5,1,10,4), nrow = 2, ncol = 3, byrow =
TRUE)
> t = M %*% t(M)
> t
```

It produces the following result –

```
      [,1] [,2]
[1,]    65   82
[2,]    82  117
```

Conditional statements

R if statement

The syntax of `if statement` is:

```
if (test_expression) {
statement
```

```
}
```

If the `test_expression` is `TRUE`, then the statement gets executed. But if it's `FALSE`, then nothing happens.
Here, `test_expression` can be a logical or numeric vector, but only the first element is taken into consideration.
In the case of numeric vectors, zero is taken as `FALSE`, rest as `TRUE`.

Flowchart of if statement

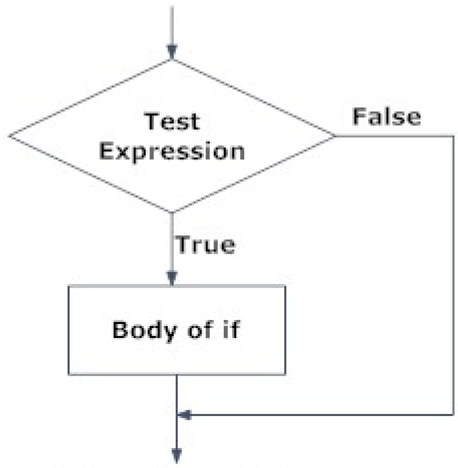


Fig: Operation of if statement

```
> x <- 5
> if (x > 0) {
   print("Positive number")
}
[1] "Positive number"
```

if…else statement

The syntax of the `if…else` statement is:

```
if (test_expression) {
statement1
} else {
statement2
}
```

The else part is optional and is only evaluated if `test_expression` is `FALSE`.
It is **important** to note that else must be in the same line as the closing brackets of the if statement.

Flowchart of if…else statement

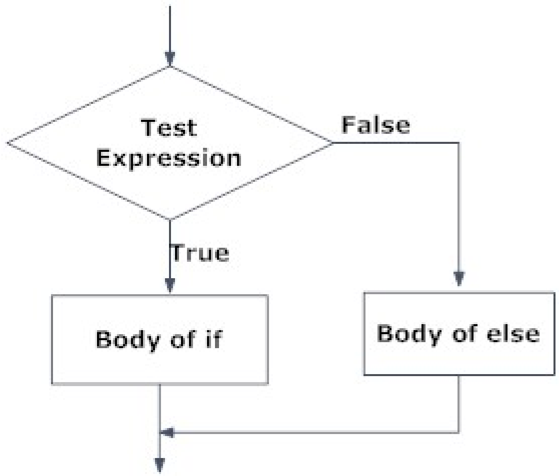


Fig: Operation of if...else statement

```
> x <- -5
> if (x > 0)
{
print("Non-negative number")
} else {
print("Negative number")
}
[1] "Negative number"
```

if…else Ladder

The if…else ladder (if…else…if) statement allows you to execute a block of code among more than 2 alternatives
The syntax of the `if…else` statement is:

```
if ( test_expression1 )
{
statement1
} else if ( test_expression2 ) {
statement2
} else if ( test_expression3 )
{
Statement 3
} else {
Statement 4
}
```

Only one statement will get executed depending upon the `test_expressions`.

```
x <- 0
if (x < 0) {
print("Negative number")
} else if (x > 0) {
print("Positive number")
} else
print("Zero")
[1] "Zero"
```

R Switch Statement

The R If Else Statement allows us to choose between `TRUE` or `FALSE`, and when there are more than two options, we simply use Nested If Else statement. Say, What if we have 12 alternatives to choose?, if we use Nested If-Else in this situation, programming logic will be difficult to understand. In R Programming, Switch statement and Else if statement can handle these type of problems effectively.

R Switch syntax

```
switch (Expression, "Option 1", "Option 2", "Option 3", ....., "Option N")
```

```
switch (Expression,
          "Option 1" = Execute these statements when the expression result match Option 1,
          "Option 2" = Execute these statements when the expression result match Option 2,
            "Option 3" = When the expression result match Option 3,Execute these statements,
        ....
        ....
            "Option N" = When the expression result match Option N,Execute these statements,
        Default Statements
)
```

- The expression value should be either integer or characters (We can write the expression as n/2 also but the result should be an integer or convertible integers).
- The R Switch statement allows us to add the default statement. If the Expression value or the `Index_Position` is not matching with any of the case statements then the default statements will be executed.
- If there is more than one match, the first matching statement will be returned.

```
switch(3,
       "Learn",
       "R Programming",
       "Tutorial",
```

```
        "Programming"
)
[1]  Tutorial
```

R for Loop

Loops are used in programming to repeat a specific block of code. The most commonly used loop is the `for` loop. It iterates over an index-provided as a vector and performs some operations.

Syntax of for loop

```
for(val in sequence)
{
statement
}
```

Here, the sequence is a vector and `val` takes on each of its value during the loop. In each iteration, the statement is evaluated.

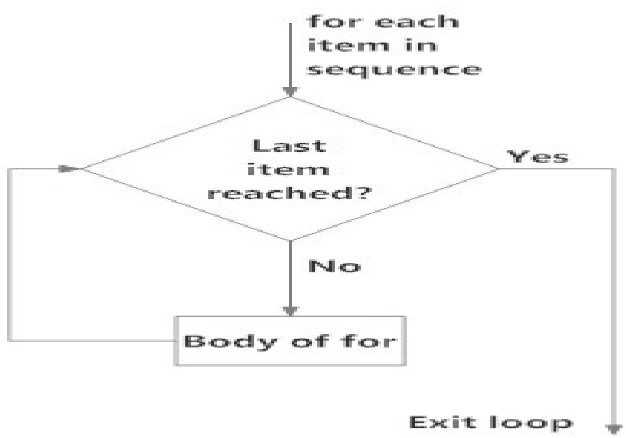


Fig: operation of for loop

```
> for ( i in 1:10 )
 {
      print (i)
   }
[1]     1
[1]     2
[1]     3
[1]     4
[1]     5
```

```
[1]     6
[1]     7
[1]     8
[1]     9
[1]     10
```

R while Loop

Loops are used in programming to repeat a specific block of code. In this, you will learn to create a while loop in R programming.

Syntax of while loop

```
while (test_expression)
{
statement
}
```

Here, `test_expression` is evaluated and the body of the loop is entered if the result is `TRUE`.
The statements inside the loop are executed and the flow returns to evaluate the `test_expression` again.
This is repeated each time until `test_expression` evaluates to `FALSE`, in which case, the loop exits.

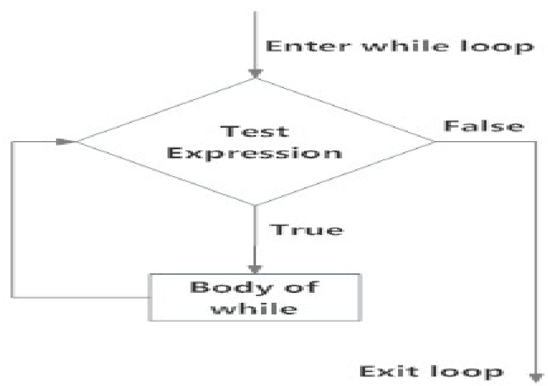


Fig: operation of while loop

```
> i <- 1
> while (i < 6)
{
 print(i)
i = i+1
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Break statement

A break statement is used inside a loop `(for, while)` to stop the iterations and flow the control outside of the loop.

In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

```
if (test_expression)
{
break
}

> x <- 1:5
> for (val in x)
{
if (val == 3)
{
break
}
print(val)
}
[1] 1
[1] 2
```

Function

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions. In R, a function is an object to the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions. The function, in turn, performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Syntax

An R function is created by using the keyword `function`. The basic syntax of an R function definition is as follows −

```
function_name <- function(arg_1, arg_2, ...)
{
   Function body
}
```

The different parts of a function are −

- **Function Name** − This is the actual name of the function. It is stored in the R environment as an object with this name.

- **Arguments** − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also, arguments can have default values.
- **Function Body** − The function body contains a collection of statements that define what the function does.
- **Return Value** − The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred to as **user-defined** functions.

Built-in Function

Simple examples of in-built functions are **`seq()`**, **`mean()`**, **`max()`**, **`sum(x)`** and **`paste(...)`** etc. They are directly called by user-written programs. You can refer the most widely used R functions.

```
# Create a sequence of numbers from 32 to 44.
print(seq(32,44))
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44

# Find mean of numbers from 25 to 82.
print(mean(25:82))
[1] 53.5

# Find sum of numbers from 41 to 68.
print(sum(41:68))
[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used as the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
                     new.function <- function(a)
           {
             for(i in 1:a) {
                b <- i^2
                print(b)
             }
          }
```

Calling a Function

```
# Create a function to print squares of numbers in sequence.
  new.function <- function(a)
      {
        for(i in 1:a) {
```

```
            b <- i^2
            print(b)
         }
      }
      # Call the function new.function supplying 6 as an argument.
               new.function(6)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an argument.
new.function <- function()
{
   for(i in 1:5) {
      print(i^2)
   }
}

# Call the function without supplying an argument.
new.function()
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
new.function <- function(a, b, c)
{
   result <- a * b + c
   print(result)
}

# Call the function by the position of arguments.
new.function(5, 3, 11)
```

```
# Call the function by names of the arguments.
new.function(a = 11, b = 5, c = 3)

[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non-default result.

```
# Create a function with arguments.
new.function <- function(a = 3, b = 6)
 {
   result <- a * b
   print(result)
}

# Call the function without giving any argument.
new.function()

# Call the function with giving new values of the argument.
new.function(9, 5)
[1] 18
[1] 45
```

R Programming
Module 2: Data Importing in R

Learning objectives

1. In this module, we are going to talk about importing common data format that we often encounter, such as Excel, or Text data.
2. Import from the file system or a URL.
3. Analyzing the CSV File.
4. You will get to know about packages to read flat files with readr and packages to read excel data.
5. How to import and work with data coming from the web.

Importing data from flat files with utils

Data comes in different formats and from different sources. Lots of data come in the form of flat files: simple tabular text files, we have to import all common formats of flat file data with base R functions.

Flat file

Flat files come in the form of .csv , .txt. To read .csv files we need the "utils " package(by default it is loaded in Rstudio). In CSV file, fields are separated by a 'comma' and hence it is .csv In R, we can read data from files stored outside the R environment. We can also write data into files that will be stored and accessed by the operating system. R can read and write into various file formats like csv, excel etc.

- R base functions for importing data: `read.table(), read.delim(), read.csv(), read.csv2()`
- Reading a local file
- Reading a file from the internet

R base functions for importing data

The R base function **`read.table`**`()` is a general function that can be used to read a file in table format. The data will be imported as a data frame.

- **`read.csv`**`()`: for reading **"comma separated value"** files (".csv").
- **`read.csv2`**`()`: variant used in countries that use a comma "," as the decimal point and a semicolon ";" as field separators.
- **`read.delim`**`()`: for reading *"tab-separated value"* files (".txt"). By default, point (".") is used as decimal points.

The simplified format of these functions are as follows

```
# Read tabular data into R
read.table(file, header = FALSE, sep = "", dec = ".")

# Read "comma separated value" files (".csv")
read.csv(file, header = TRUE, sep = ",", dec = ".", ...)

# Or use read.csv2: variant used in countries that
# use a comma as decimal point and a semicolon as field separator.
read.csv2(file, header = TRUE, sep = ";", dec = ",", ...)

# Read TAB delimited files
read.delim(file, header = TRUE, sep = "\t", dec = ".", ...)
```

- **file**: the path to the file containing the data to be imported into R.
- **sep**: the field separator character. "\t" is used for tab-delimited file.
- **header**: logical value. If TRUE, **read.table()** assumes that your file has a header row, so row 1 is the name of each column. If that's not the case, you can add the argument **header = FALSE**.
- **dec**: the character used in the file for decimal points.

CSV file: we will learn to read data from a csv file and then write data into a csv file. The file should be present in the current working directory so that R can read it. Of course, we can also set our own directory and read files from there.

Getting and Setting the Working Directory

You can check which directory the R workspace is pointing to using the `getwd()` function. You can also set a new working directory using `setwd()` function.

```
# Get and print the current working directory.
print(getwd())

# Set the current working directory.
setwd("/web/com")
# Get and print the current working directory.
print(getwd())
```

Input as CSV File

The csv file is a text file in which the values in the columns are separated by a comma. Let's consider the following data present in the file named input.csv.

You can create this file using windows notepad by copying and pasting this data. Save the file as `input.csv` using the "Save As" All Files (*.*) option in notepad.

```
id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Michelle,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
5,Gary,843.25,2015-03-27,Finance
6,Nina,578,2013-05-21,IT
7,Simon,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance
```

Reading a CSV File

Following is a simple example of `read.csv()` function to read a CSV file available in your current working directory –

```
data <- read.csv("input.csv")
print(data)
```

When we execute the above code, it produces the following result –

	id,	name,	salary,	start_date,	dept
1	1	Rick	623.30	2012-01-01	IT
2	2	Dan	515.20	2013-09-23	Operations
3	3	Michelle	611.00	2014-11-15	IT
4	4	Ryan	729.00	2014-05-11	HR
5	NA	Gary	843.25	2015-03-27	Finance

6	6	Nina	578.00	2013-05-21	IT
7	7	Simon	632.80	2013-07-30	Operations
8	8	Guru	722.50	2014-06-17	Finance

Analyzing the CSV File

By default, the read.csv() function gives the output as a data frame. This can be easily checked as follows. Also, we can check the number of columns and rows.

```
data <- read.csv("input.csv")

print(is.data.frame(data))
print(ncol(data))
print(nrow(data))
```

When we execute the above code, it produces the following result −

```
[1] TRUE
[1] 5
[1] 8
```

Packages to read flat files with readr

The goal of the readr package is to provide a fast and friendly way to read rectangular data (like `csv` and `tsv`). It is designed to flexibly parse many types of data found in the wild, while still cleanly failing when data unexpectedly changes.

- Functions for reading txt|csv files: **`read_delim`**`()`, **`read_tsv`**`()`, **`read_csv`**`()`, **`read_csv2`**`()`
- Reading a file
    - Reading a local file
    - Reading a file from the internet
    - In the case of parsing problems
- Specify column types
- Reading lines from a file: **`read_lines`**`()`
- Read the whole file: **`read_file`**`()`

`readr` supports several file formats with several `read_` functions:

- `read_csv()`: comma separated (CSV) files
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_table()`: tabular files where columns are separated by white-space.

Installation

The easiest way to get readr is to install the whole tidyverse:

```
install.packages("tidyverse")
```

Alternatively, install just readr:

```
install.packages("readr")
```

Or the the development version from GitHub:

```
install.packages("devtools")
devtools::install_github("tidyverse/readr")
```

In many cases, these functions will just work: you supply the path to a file and you get a tibble back. The following example loads a sample file bundled with readr:

```
mtcars <- read_csv(readr_example("mtcars.csv"))
#> Parsed with column specification:
#> cols(
#>   mpg = col_double(),
#>   cyl = col_double(),
#>   disp = col_double(),
#>   hp = col_double(),
#>   drat = col_double(),
#>   wt = col_double(),
#>   qsec = col_double(),
#>   vs = col_double(),
#>   am = col_double(),
#>   gear = col_double(),
#>   carb = col_double()
#> )
```

Note that readr prints the column specification. This is useful because it allows you to check that the columns have been read in as you expect, and if they haven't, you can easily copy and paste into a new call:

```
mtcars <- read_csv(readr_example("mtcars.csv"), col_types =
  cols(
    mpg = col_double(),
    cyl = col_integer(),
    disp = col_double(),
    hp = col_integer(),
    drat = col_double(),
    vs = col_integer(),
    wt = col_double(),
    qsec = col_double(),
    am = col_integer(),
    gear = col_integer(),
    carb = col_integer()
  )
)
```

Packages to read excel data

Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format. R can read directly from these files using some excel specific packages. Few such packages are - XLConnect, xlsx, gdata etc. We will be using the xlsx package. R can also write into excel file using this package.

- Copying data from Excel and import into R
- Importing Excel files into R using `readxl` package
- Importing Excel files using `xlsx` package
- The `readxl` package is used to read and import .xlsx excel data into the R.

```
# Use readxl package to read xls|xlsx
library("readxl")
my_data <- read_excel("my_file.xlsx")
# Use xlsx package
library("xlsx")
my_data <- read.xlsx("my_file.xlsx")
```

Input as xlsx File

Open Microsoft excel. Copy and paste the following data in the worksheet named as sheet1.

```
id    name       salary    start_date      dept
1     Rick      623.3     1/1/2012        IT
2     Dan       515.2      9/23/2013      Operations
3     Michelle  611        11/15/2014     IT
4     Ryan      729        5/11/2014      HR
5     Gary      43.25     3/27/2015       Finance
6     Nina      578         5/21/2013     IT
7     Simon     632.8      7/30/2013      Operations
8     Guru      722.5      6/17/2014      Finance
```

Also, copy and paste the following data to another worksheet and rename this worksheet to "city".

```
name       city
Rick       Seattle
Dan         Tampa
Michelle    Chicago
Ryan       Seattle
Gary       Houston
Nina       Boston
Simon      Mumbai
Guru       Dallas
```

Save the Excel file as "input.xlsx". You should save it in the current working directory of the R workspace.

Reading the Excel File

The input.xlsx is read by using the `read.xlsx()` function as shown below. The result is stored as a data frame in the R environment.

```
# Read the first worksheet in the file input.xlsx
data <- read.xlsx("input.xlsx", sheetIndex = 1)
print(data)
```

When we execute the above code, it produces the following result −

```
      id,    name,        salary,      start_date,    dept
1      1     Rick          623.30      2012-01-01     IT
2      2     Dan           515.20      2013-09-23     Operations
3      3     Michelle     611.00      2014-11-15      IT
4      4     Ryan         729.00      2014-05-11      HR
5     NA     Gary          843.25     2015-03-27      Finance
6      6     Nina          578.00     2013-05-21      IT
7      7     Simon        632.80      2013-07-30      Operations
8      8     Guru         722.50     2014-06-17       Finance
```

Importing data from the web

Many websites provide data for consumption by its users. For example the World Health Organization (WHO) provides reports on health and medical information in the form of CSV, txt and XML files.

Using R programs, we can programmatically extract specific data from such websites. Some packages in R which are used to scrape data from the web are − `"RCurl"`, `″XML"`, and `"stringr"`. They are used to connect to the URL's, identify required links for the files and download them to the local environment.

Install R Packages

The following packages are required for processing the URL and links to the files. If they are not available in your R Environment, you can install them using the following commands.

```
install.packages("RCurl")
install.packages("XML")
install.packages("stringr")
install.packages("plyr")
```

Input Data

We will visit the URL weather data and download the CSV files using R for the year 2015.

Example:

We will use the function `getHTMLLinks()` to gather the URLs of the files. Then we will use the function `download.file()` to save the files to the local system.

As we will be applying the same code again and again for multiple files, we will create a function to be called multiple times. The filenames are passed as parameters in form of a R list object to this function.

```
# Read the URL.
url <- "http://www.geos.ed.ac.uk/~weather/jcmb_ws/"

# Gather the html links present in the webpage.
links <- getHTMLLinks(url)
# Identify only the links which point to the JCMB 2015 files.
filenames <- links[str_detect(links, "JCMB_2015")]

# Store the file names as a list.
filenames_list <- as.list(filenames)

# Create a function to download the files by passing the URL and filename list.
download csv <- function (main url,filename) {
   filedetails <- str_c(main url,filename)
   download.file(filedetails,filename)
}

# Now apply the l_ply function and save the files into the current R working directory.
l_ply(filenames,downloadcsv,mainurl="http://www.geos.ed.ac.uk/~weather/jcmb_ws/")
```

R - XML Files

XML is a file format which shares both the file format and the data on the World Wide Web, intranets, and elsewhere using standard ASCII text. It stands for Extensible Markup Language (XML). Similar to HTML it contains markup tags. But unlike HTML where the markup tag describes the structure of the page, in xml markup tags describe the meaning of the data contained within the file.

You can read an xml file in R using the `"XML"` package. This package can be installed using following command.

```
install.packages("XML")
```

Input Data

Create an XMl file by copying the below data into a text editor like notepad. Save the file with a .xml extension and choosing the file type as All Files (*.*).

```
<RECORDS>
   <EMPLOYEE>
      <ID>1</ID>
      <NAME>Rick</NAME>
      <SALARY>623.3</SALARY>
```

```
   <STARTDATE>1/1/2012</STARTDATE>
   <DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
   <ID>2</ID>
   <NAME>Dan</NAME>
   <SALARY>515.2</SALARY>
   <STARTDATE>9/23/2013</STARTDATE>
   <DEPT>Operations</DEPT>
</EMPLOYEE>

<EMPLOYEE>
   <ID>3</ID>
   <NAME>Michelle</NAME>
   <SALARY>611</SALARY>
   <STARTDATE>11/15/2014</STARTDATE>
   <DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
   <ID>4</ID>
   <NAME>Ryan</NAME>
   <SALARY>729</SALARY>
   <STARTDATE>5/11/2014</STARTDATE>
   <DEPT>HR</DEPT>
</EMPLOYEE>

<EMPLOYEE>
   <ID>5</ID>
   <NAME>Gary</NAME>
   <SALARY>843.25</SALARY>
   <STARTDATE>3/27/2015</STARTDATE>
   <DEPT>Finance</DEPT>
</EMPLOYEE>

  <EMPLOYEE>
   <ID>6</ID>
   <NAME>Nina</NAME>
   <SALARY>578</SALARY>
   <STARTDATE>5/21/2013</STARTDATE>
   <DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
   <ID>7</ID>
```

```
        <NAME>Simon</NAME>
        <SALARY>632.8</SALARY>
        <STARTDATE>7/30/2013</STARTDATE>
        <DEPT>Operations</DEPT>
    </EMPLOYEE>

        <EMPLOYEE>
        <ID>8</ID>
        <NAME>Guru</NAME>
        <SALARY>722.5</SALARY>
        <STARTDATE>6/17/2014</STARTDATE>
        <DEPT>Finance</DEPT>
    </EMPLOYEE>
</RECORDS>
```

Reading XML File

The xml file is read by R using the function xmlParse(). It is stored as a list in R.

```
# Load the package required to read XML files.
library("XML")

# Also load the other required package.
library("methods")

# Give the input filename to the function.
result <- xmlParse(file = "input.xml")

# Print the result.
print(result)
```

When we execute the above code, it produces the following result −

```
1
Rick
623.3
1/1/2012
IT

2
Dan
515.2
9/23/2013
Operations

3
Michelle
611
```

```
11/15/2014
IT

4
Ryan
729
5/11/2014
HR

5
Gary
843.25
3/27/2015
Finance

6
Nina
578
5/21/2013
IT

7
Simon
632.8
7/30/2013
Operations

8
Guru
722.5
6/17/2014
Finance
```

Get Number of Nodes Present in XML File

```
# Load the packages required to read XML files.
library("XML")
library("methods")

# Give the input file name to the function.
result <- xmlParse(file = "input.xml")

# Extract the root node form the xml file.
rootnode <- xmlRoot(result)

# Find the number of nodes in the root.
root size <- xml Size(root node)
```

```
# Print the result.
print(rootsize)
```

When we execute the above code, it produces the following result −

```
output
[1] 8
```

Let's look at the first record of the parsed file. It will give us an idea of the various elements present in the top level node.

```
# Load the packages required to read XML files.
library("XML")
library("methods")

# Give the input filename to the function.
result <- xmlParse(file = "input.xml")

# Extract the root node form the xml file.
rootnode <- xmlRoot(result)

# Print the result.
print(rootnode[1])
```

When we execute the above code, it produces the following result −

```
$EMPLOYEE
   1
   Rick
   623.3
   1/1/2012
   IT

attr(,"class")
[1] "XML Internal NodeList" "XMLNodeList"
```

Get Different Elements of a Node

```
# Load the packages required to read XML files.
library("XML")
library("methods")

# Give the input filename to the function.
result <- xmlParse(file = "input.xml")

# Extract the root node form the xml file.
rootnode <- xmlRoot(result)
```

```
# Get the first element of the first node.
print(rootnode[[1]][[1]])

# Get the fifth element of the first node.
print(rootnode[[1]][[5]])

# Get the second element of the third node.
print(rootnode[[3]][[2]])
```

When we execute the above code, it produces the following result −

```
1
IT
Michelle
```

XML to Data Frame

To handle the data effectively in large files we read the data in the xml file as a data frame. Then process the data frame for data analysis.

```
# Load the packages required to read XML files.
library("XML")
library("methods")

# Convert the input xml file to a data frame.
xmltodataframe <- xmlToDataFrame("input.xml")
print(xmltodataframe)
```

When we execute the above code, it produces the following result −

```
      ID    NAME          SALARY      STARTDATE          DEPT
1      1    Rick          623.30      2012-01-01         IT
2      2    Dan           515.20      2013-09-23         Operations
3      3    Michelle      611.00      2014-11-15         IT
4      4    Ryan          729.00      2014-05-11         HR
5     NA    Gary          843.25      2015-03-27         Finance
6      6    Nina          578.00      2013-05-21         IT
7      7    Simon         632.80      2013-07-30         Operations
8      8    Guru          722.50      2014-06-17         Finance
```

As the data is now available as a dataframe we can use data frame related functions to read and manipulate the file.

R Programming
Module 3: Data Manipulation in R

Learning objectives

1. How to explore Raw data in R.
2. The process of cleaning data in R.
3. How to handle dates, string, and missing values.
4. How to use functions that perform mostly used data manipulation operations.
5. Describe the purpose of the tidyr packages.

Introduction and exploring raw data

Exploring Raw data

This module will give you an overview of the process of data cleaning with R, then walk you through the basics of exploring raw data.

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

#Loading BOSTON data
> Boston

# View its dimensions(ROWS AND COLUMNS)
> dim(Boston)
[1] 506  14

# Look at column names
> names(Boston)

[1]  "crim"     "zn"        "indus"    "chas"     "nox"      "rm"
"age"     "dis"     "rad"
[10] "tax"     "ptratio" "black"   "lstat"   "medv"

# Look at the class
> class(Boston)
[1] "data.frame"
```

Viewing the structure of your data

```
#view structure of Boston
> str(Boston)

'data.frame'   : 506 obs. of  14 variables:
 $ crim   : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
 $ zn     : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5 ...
 $ indus   : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87
...
 $ chas   : num  0 0 0 0 0 0 0 0 0 0 ...
 $ nox    : num  0.538 0.469 0.469 0.458 0.458 0.458 0.524 ...
 $ rm     : num  6.58 6.42 7.18 7 7.15 ...
 $ age     : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100
...
 $ dis    : num  4.09 4.97 4.97 6.06 6.06 ...
 $ rad    : num  1 2 2 3 3 3 5 5 5 5 ...
 $ tax    : num  296 242 242 222 222 222 311 311 311 311 ...
 $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2
...
```

```
 $ black  : num  397 397 393 395 397 ...
 $ lstat  : num  4.98 9.14 4.03 2.94 5.33 ...
 $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18
```

`glimpse()` *function*

The glimpse function from dplyr is a slightly cleaner alternative to `str()`. `str()` and `glimpse()` give you a preview of your data, which may reveal issues with the way columns are labelled, how variables are encoded, etc.

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

You can use the `summary()` command to get a better feel for how your data is distributed, which may reveal unusual or extreme values, unexpected missing data, etc. For numeric variables, this means looking at means, quartiles (including the median), and extreme values. For character or factor variables, you may be curious about the number of times each value appears in the data (i.e. counts), which `summary()` also reveals.

```
#view summary of Boston
> summary(Boston)
```

Looking at the data

The most basic way to look at data in R is by printing it to the console. The `print()` command is not even necessary; you can just type the name of the object. The downside to this option is that R will attempt to print the entire dataset, which can be a nuisance if the dataset is too large. One way around this is to use the `head()` and `tail()` commands, which only display the first and last 6 rows of data, respectively. You can view more (or fewer) rows by providing as a second argument to the function the number of rows you wish to view. These functions provide a useful method for quickly getting a sense of your data without overly cluttering the console.

```
# View the first 6 rows of data
> head(Boston)
```

```
     crim zn indus  chas   nox  rm    age   dis rad tax ptratio  black lstat medv
1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2
6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21 28.7
```

```
# View the last 6 rows of data()
> tail(Boston)
```

```
       crim zn indus  chas  nox    rm   age   dis  rad tax  ptratio  black lstat medv
501 0.22438  0  9.69    0 0.585 6.027 79.7 2.4982  6  391   19.2   396.90 14.33 16.8
502 0.06263  0 11.93    0 0.573 6.593 69.1 2.4786  1  273   21.0   391.99  9.67 22.4
503 0.04527  0 11.93    0 0.573 6.120 76.7 2.2875   1 273    21.0 396.90  9.08 20.6
```

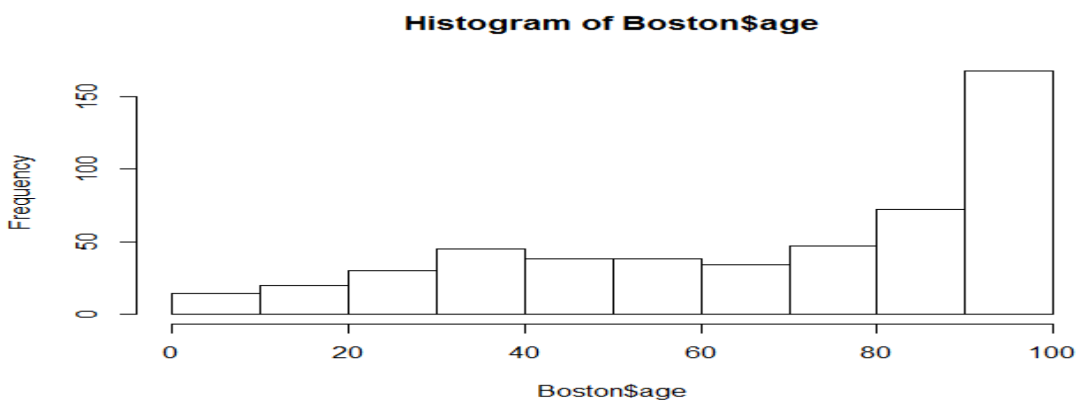
```
504 0.06076  0 11.93    0 0.573 6.976 91.0 2.1675   1 273    21.0 396.90  5.64 23.9
505 0.10959  0 11.93    0 0.573 6.794 89.3 2.3889   1 273    21.0 393.45  6.48 22.0
506 0.04741  0 11.93    0 0.573 6.030 80.8 2.5050   1 273    21.0 396.90  7.88 11.9
```

```
#to get 13 number of rows from the bottom in the given data set
> tail(Boston,n=13)
#to get 12 number of rows from the top in the given data
> head( Boston,n=12)
```

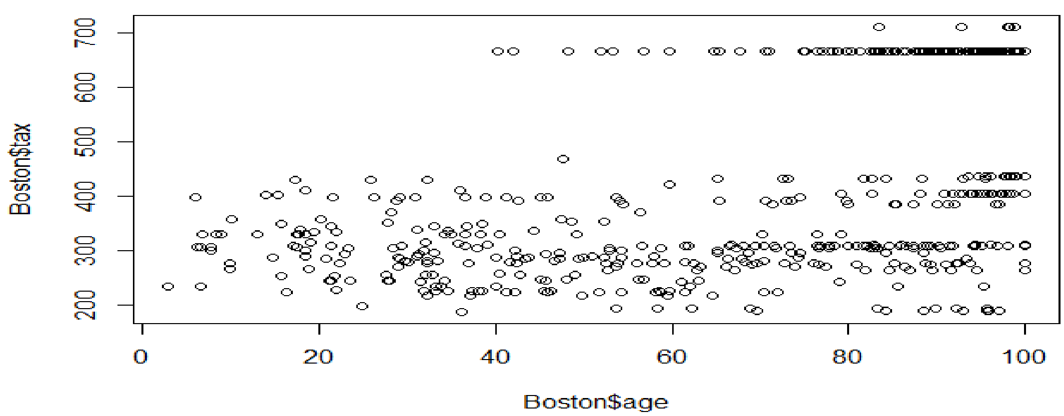
Visualizing data

Visualization is one of the important things to do. There are many ways to visualize data. Plotting them is the best among them.

```
# View histogram
> hist(Boston$age)
```




```
# View plot of two variables
> plot(Boston$age,Boston$tax)
```



Tidying data with `tidyr`

Tidying data

We'll focus on `tidyr`, a package that provides a bunch of tools to help tidy up your messy datasets. `tidyr` is a member of the core `tidyverse`.

```
library(tidyverse)
```

You can represent the same underlying data in multiple ways. The following example shows the same data organized in four different ways. Each dataset shows the same values of four variables, country, year, population, and cases, but each dataset organizes the values in a different way:

```
> table1
#> # A tibble: 6 × 4
     country      year    cases   population
       <chr>     <int>   <int>       <int>
   1  Afghanistan   1999     745    19987071
   2  Afghanistan   2000    2666    20595360
   3  Brazil        1999   37737   172006362
   4  Brazil        2000   80488   174504898
   5  China         1999  212258  1272915272
   6  China         2000  213766  1280428583

> table2
#> # A tibble: 12 × 4
       country        year      type           count
        <chr>         <int>    <chr>           <int>
   1 Afghanistan     1999     cases              745
   2 Afghanistan     1999     population    19987071
   3 Afghanistan     2000     cases             2666
   4 Afghanistan     2000     population    20595360
   5 Brazil          1999     cases            37737
   6 Brazil          1999     population   172006362
   ... with 6 more rows

> table3
# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583

# Spread across two tibbles
> table4a #cases
# A tibble: 3 x 3
```

```
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766

> table4b #population
# A tibble: 3 x 3
  country          `1999`     `2000`
* <chr>             <int>      <int>
1 Afghanistan   19987071   20595360
2 Brazil       172006362  174504898
3 China       1272915272 1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the `tidyverse`.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

These three rules are interrelated because it's impossible to only satisfy two of the three. That interrelationship leads to an even simpler set of practical instructions:

1. Put each dataset in a tibble.
2. Put each variable in a column.

In this example, only `table1` is tidy. It's the only representation where each column is a variable.

Why ensure that your data is tidy? There are two main advantages:

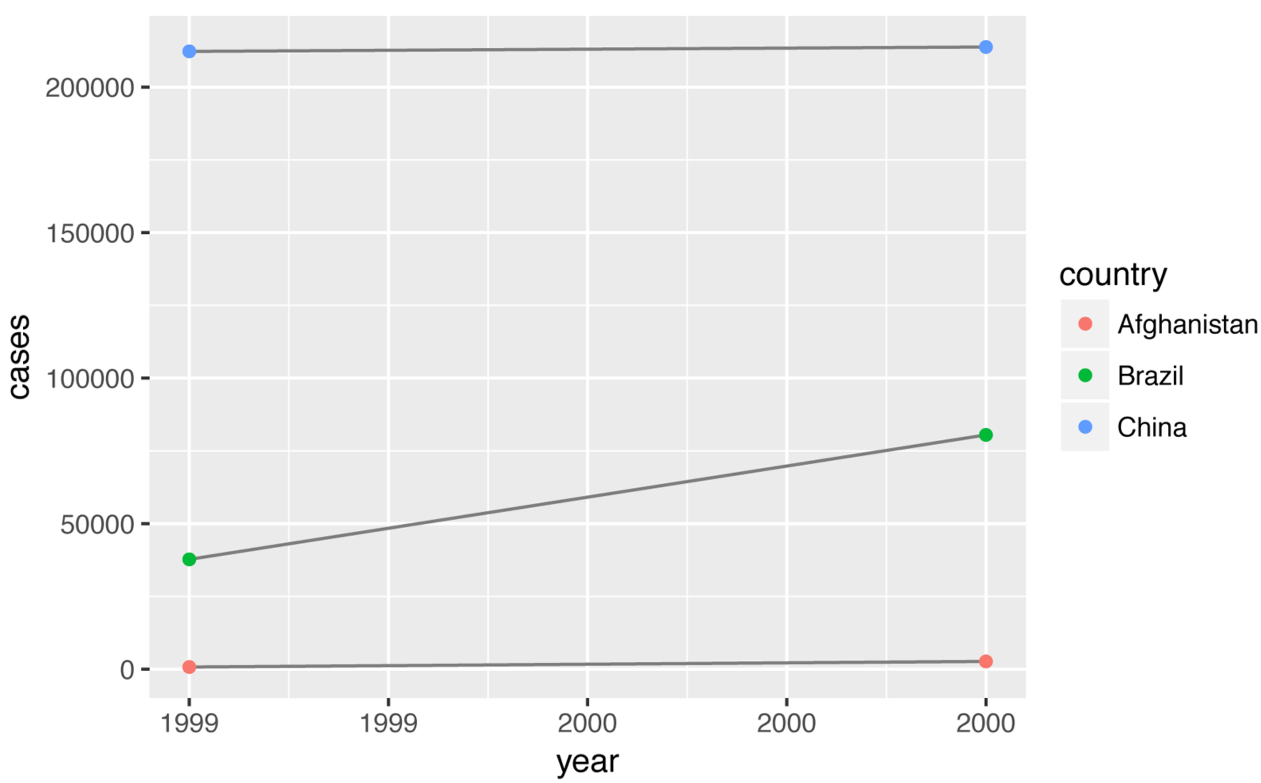
- There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

- There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine. Most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

`dplyr`, `ggplot2`, and all the other packages in the tidyverse are designed to work with tidy data. Here are a couple of small examples showing how you might work with `table1` :

```
# Compute rate per 10,000
table1 %>%
mutate(rate = cases / population * 10000)
```

```
#> # A tibble: 6 × 5
#>       country          year      cases     population        rate
#>       <chr>            <int>     <int>       <int>           <dbl>
#> 1 Afghanistan          1999        745     19987071          0.373
#> 2 Afghanistan          2000       2666     20595360          1.294
#> 3 Brazil               1999      37737    172006362          2.194
#> 4 Brazil               2000      80488    174504898          4.612
#> 5 China                1999     212258   1272915272          1.667
#> 6 China                2000     213766   1280428583          1.669

# Compute cases per year
table1 %>%
count(year, wt = cases)
#> # A tibble: 2 × 2
#>   year     n
#><int>    <int>
#> 1 1999 250740
#> 2 2000 296920

# Visualize changes over time
library(ggplot2)
ggplot(table1, aes(year, cases)) +
geom_line(aes(group = country), color = "grey 50") +
geom_point(aes(color = country))
```



- **Spreading and Gathering**

  The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

  - Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a lot of time working with data.
  - Data is often organized to facilitate some use other than analysis. For example, data is often organized to make entry as easy as possible.

  This means for most real analyses, you'll need to do some tidying.The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

  - One variable might be spread across multiple columns.
  - One observation might be scattered across multiple rows.

  Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in `tidyr: gather()` and `spread().`

- **Gathering**

  A common problem is a dataset where some of the column names are not names of variables, but the values of a variable. Take table `4a`; the column names 1999 and 2000 represent values of the year variable, and each row represents two observations, not one:

  ```
  table 4a
  #> # A tibble: 3 × 3
  #>      country       `1999`   `2000`
  #> *   <chr>          <int>     <int>
  #> 1  Afghanistan       745      2666
  #> 2  Brazil          37737     80488
  #> 3  China          212258    213766
  ```

  To tidy a dataset like this, we need to gather those columns into a new pair of variables. To describe that operation we need three parameters:

  - The set of columns that represent values, not variables. In this example, those are the columns 1999 and 2000.
  - The name of the variable whose values form the column names. I call that the key, and here it is the year.
  - The name of the variable whose values are spread over the cells.I call that value, and here it's the number of cases.

Together those parameters generate the call to `gather()` :

```
table 4a %>%
gather(`1999`, `2000`, key = "year", value = "cases")
#> # A tibble: 6 × 3
#>    country          year       cases
#>    <chr>            <chr>      <int>
#> 1 Afghanistan       1999        745
#> 2 Brazil            1999        37737
#> 3 China             1999        212258
#> 4 Afghanistan       2000        2666
#> 5 Brazil            2000        80488
#> 6 China             2000        213766
```

The columns to gather are specified with `dplyr::select()` style notation. Here there are only two columns, so we list them individually. Note that “`1999`” and “`2000`” are non syntactic names so we have to surround them in backticks.

In the final result, the gathered columns are dropped, and we get new key and value columns. Otherwise, the relationships between the original variables are preserved. We can use `gather()` to tidy `table 4b` in a similar fashion. The only difference is the variable stored in the cell values:

```
table 4b %>%
gather(`1999`, `2000`, key = "year", value = "population")
#> # A tibble: 6 × 3
#>       country          year        population
#>        <chr>           <chr>          <int>
#> 1 Afghanistan          1999       19987071
#> 2      Brazil          1999       172006362
#> 3       China          1999       1272915272
#> 4 Afghanistan          2000       20595360
#> 5      Brazil          2000       174504898
#> 6       China          2000       1280428583
```

Figure: Gathering table 4 into a tidy form

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`:

```
tidy4a <- table4a %>%
gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
#> Joining, by = c("country", "year")
#> # A tibble: 6 × 4
#>      country        year     cases        population
#>      <chr>         <chr>     <int>          <int>
#> 1 Afghanistan       1999       745          19987071
#> 2      Brazil       1999     37737         172006362
#> 3       China       1999    212258        1272915272
#> 4 Afghanistan       2000      2666          20595360
#> 5      Brazil       2000     80488         174504898
#> 6          China   2000   213766  1280428583
```

- **Spreading**
  Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. For example, take `table2` —an observation is a `country` in a `year`, but each observation is spread across two rows:

```
table2
#> # A tibble: 12 × 4
#>      country         year     type          count
#>      <chr>           <int>    <chr>         <int>
#> 1 Afghanistan        1999     cases             745
#> 2 Afghanistan        1999     population   19987071
#> 3 Afghanistan        2000     cases            2666
```

```
#> 4 Afghanistan        2000     population     20595360
#> 5      Brazil        1999     cases             37737
|#> 6     Brazil        1999     population    172006362
#> # ... with 6 more rows
```

To tidy this up, we first analyze the representation in a similar way to `gather()`. This time, however, we only need two parameters:

- The column that contains variable names, the key column. Here, it's type.
- The column that contains values forms multiple variables, the value column. Here, it's count.

Once we've figured that out, we can use `spread()` , as shown programmatically here, and visually in figure below:

```
spread(table2, key = type, value = count)
#> # A tibble: 6 × 4
#>     country        year       cases      population
#> *  <chr>          <int>      <int>          <int>
#> 1 Afghanistan     1999         745        19987071
#> 2 Afghanistan     2000        2666        20595360
#> 3      Brazil     1999       37737       172006362
#> 4      Brazil     2000       80488       174504898
#> 5       China     1999      212258      1272915272
#> 6       China     2000      213766      1280428583
```

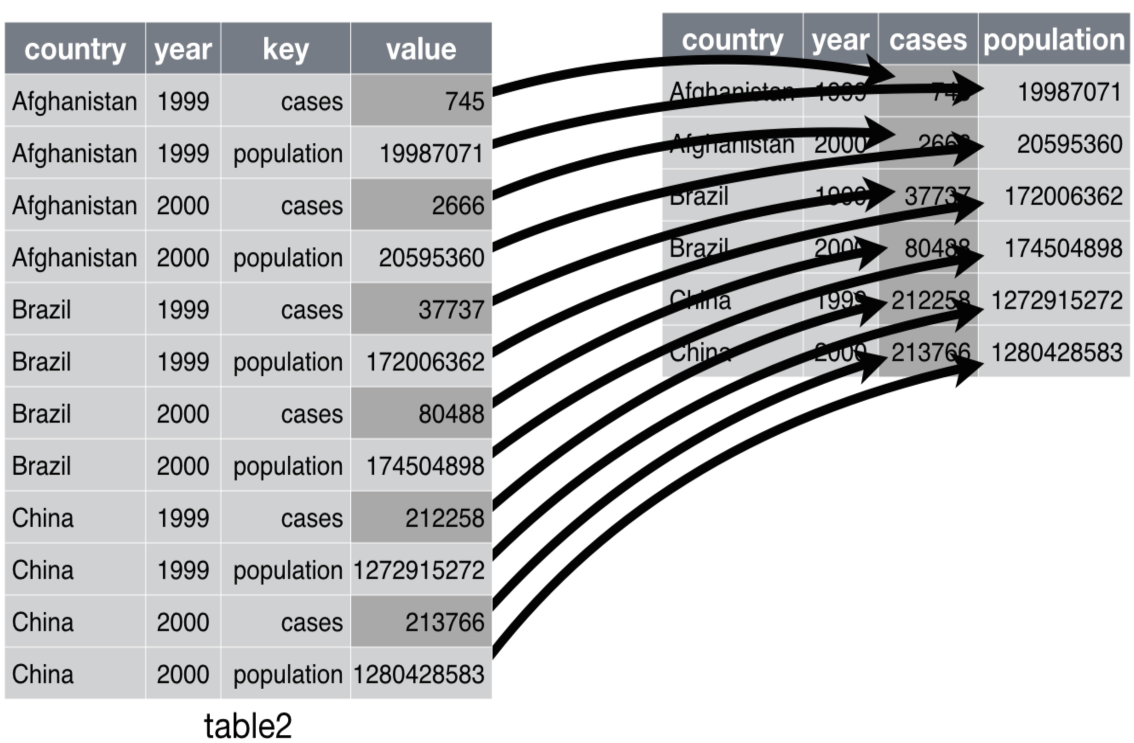


Figure: Spreading table2 makes it tidy

As you might have guessed from the common key and value arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

- **Separating and Pull**

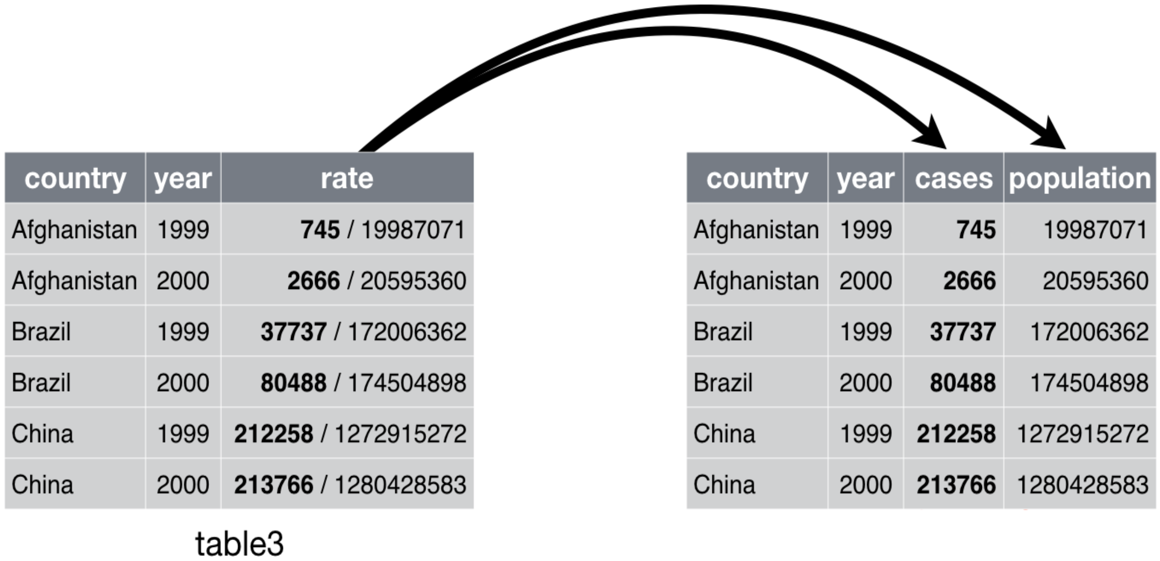
  So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table 3` has a different problem: we have one column (`rate`) that contains two variables (`cases and population`). To fix this problem, we'll need the `separate()` function. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

  `separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take table3 :

```
table3
#> # A tibble: 6 × 3
#>          country       year            rate
#> *       <chr>         <int>            <chr>
#> 1     Afghanistan      1999        745/19987071
#> 2     Afghanistan      2000        2666/20595360
#> 3          Brazil      1999        37737/172006362
#> 4          Brazil      2000        80488/174504898
#> 5           China      1999        212258/1272915272
#> 6           China      2000        213766/1280428583
```

  The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into, as shown in the figure below and the following code:

```
table3 %>%
separate(rate, into = c("cases", "population"))
#> # A tibble: 6 × 4
#>      country          year      cases       population
#> *    <chr>           <int>      <chr>          <chr>
#> 1  Afghanistan       1999         745       19987071
#> 2  Afghanistan       2000        2666       20595360
#> 3       Brazil       1999       37737      172006362
#> 4       Brazil       2000       80488      174504898
#> 5        China       1999      212258     1272915272
#> 6        China       2000      213766     1280428583
```



By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e., a character that isn't a number or letter). For example, in the preceding code, `separate()` split the values of the `rate` at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. For example, we could rewrite the preceding code as:

```
table3 %>%
separate(rate, into = c("cases", "population"), sep = "/")
```

Look carefully at the column types: you'll notice that case and population are character columns. This is the default behavior in separate() : it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using convert = TRUE :

```
table3 %>%
separate(
rate,
into = c("cases", "population"),
convert = TRUE
)
```

```
#> # A tibble: 6 × 4
#>      country      year   cases     population
#> *     <chr>       <int>  <int>      <int>
#> 1   Afghanistan   1999     745     19987071
#> 2   Afghanistan   2000    2666     20595360
#> 3        Brazil   1999   37737    172006362
#> 4        Brazil   2000   80488    174504898
#> 5         China   1999  212258   1272915272
#> 6         China   2000  213766   1280428583
```

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far left of the strings; negative values start at –1 on the far right of the strings. When using integers to separate strings, the length of sep should be one less than the number of names in. You can use this arrangement to separate the last two digits of each year. This makes this data less tidy, but is useful in other cases, as you'll see in a little bit:

```
table3 %>%
separate(year, into = c("century", "year"), sep = 2)
#> # A tibble: 6 × 4
#>      country        century     year             rate
#> *     <chr>           <chr>     <chr>            <chr>
#> 1   Afghanistan        19         99         745/19987071
#> 2   Afghanistan        20         00        2666/20595360
#> 3        Brazil        19         99       37737/172006362
#> 4        Brazil        20         00       80488/174504898
#> 5         China        19         99     212258/1272915272
#> 6         China        20         00     213766/1280428583
```

- **Unite**

  `unite()` is the inverse of `separate()` : it combines multiple columns into a single column. You'll need it much less frequently than `separate()` , but it's still a useful tool to have in your back pocket.We can use `unite()` to rejoin the century and year columns that we created in the last example. That data is saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create,and a set of columns to combine, again specified in `dplyr::select()`. The result is shown in the figure below and in the following code:

```
table5 %>%
unite(new, century, year)
#> # A tibble: 6 × 3
#>     country            new               rate
#> *   <chr>             <chr>              <chr>
#> 1   Afghanistan      19_99          745/19987071
#> 2   Afghanistan      20_00         2666/20595360
#> 3        Brazil      19_99       37737/172006362
#> 4        Brazil      20_00       80488/174504898
#> 5         China      19_99     212258/1272915272
#> 6         China      20_00     213766/1280428583
```

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	**99**	745 / 19987071
Afghanistan	20	**0**	2666 / 20595360
Brazil	19	**99**	37737 / 172006362
Brazil	20	**0**	80488 / 174504898
China	19	**99**	212258 / 1272915272
China	20	**0**	213766 / 1280428583

table6

Figure: Uniting table 5 makes it tidy

In this case we also need to use the `sep` argument. The default will place an underscore (_) between the values from different columns. Here we don't want any separator so we use "" :

```
table5 %>%
unite(new, century, year, sep = "")
#> # A tibble: 6 × 3
#>       country        new             rate
#> *     <chr>         <chr>           <chr>
#> 1   Afghanistan    1999         745/19987071
#> 2   Afghanistan    2000        2666/20595360
#> 3        Brazil    1999       37737/172006362
#> 4        Brazil    2000       80488/174504898
#> 5         China    1999     212258/1272915272
#> 6         China    2000     213766/1280428583
```

Preparing data for analysis

Types of variables in R

As in other programming languages, R is capable of storing data in many different formats. The class() function tells you what type of object you're working with.

Types of variables in R

- character: `"treatment", "123", "A"`
- numeric: `23.44, 120, NaN, Inf(NaN= not a number, Inf= Infinity)`
- integer: 4L, 1123L
- factor: `factor("Hello"), factor(8)`
- logical: `TRUE, FALSE, NA`

```
#class function is used to find the type

    > class("hello")
```

```
    [1] "character"
> class(3.844)
    [1] "numeric"
```

Common type conversions:

```
> as.character(2016)
 [1] "2016"
> as.numeric(TRUE)
 [1] 1
> as.integer(99)
 [1] 99
```

Working with dates

Dates can be a challenge to work within any programming language, but the lubridate package, working with dates in R. lubridate to help us standardize the format of dates and times in our data. A log of date conversions in lubridate are `ymd, mdy, hms, ymd_hms`, etc

```
# Load the lubridate package
>install.packages("lubridate")
> library(lubridate)

#year-month-day
> ymd("2015 August 25")
[1] "2015-08-25 UTC"

#month-day-year
> mdy("August 25, 2015")
[1] "2015-08-25 UTC"

#hour-minute-second
> hms("13:33:09")
[1] "13H 33M 9S"

#year-month-day hour-minute-second
> ymd_hms("2015/08/25 13.33.09")
"2015-08-25 13:33:09 UTC"
```

String Manipulation

the `stringr` package for string manipulation. stringr is not part of the core `tidyverse` because you don't always have textual data, so we need to load it explicitly.

```
library(tidyverse)
library(stringr)
```

- **String Basics**

  You can create strings with either single quotes or double quotes. Unlike other languages, there is no difference in behavior. I recommend always using " unless you want to create a string that contains multiple " :

```
string1 <- "This is a string"
string2 <- 'To put a "quote" inside a string, use single quotes'
```

  If you forget to close a quote, you'll see `+,` the continuation character:

```
> "This is a string without a closing quote
+
+
+ HELP I'M STUCK
If this happens to you, press Esc and try again!
```

  To include a literal single or double quote in a string you can use \ to "escape" it:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

  That means if you want to include a literal backslash, you'll need to double it up: `"\\".` Beware that the printed representation of a string is not the same as the string itself, because the printed representation shows the escapes. To see the raw contents of the string, use `writeLines()` :

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

  There are a handful of other special characters. The most common are `"\n",` newline, and `"\t",` tab, but you can see the complete list by requesting help on ?'"', or ?"'" . You'll also sometimes see strings like `"\u00b5",` which is a way of writing non-English characters that work on all platforms:

```
x <- "\u00b5"
x
#> [1] "µ"
Multiple strings are often stored in a character vector, which you can create with c() :
c("one", "two", "three")
#> [1] "one" "two" "three"
```

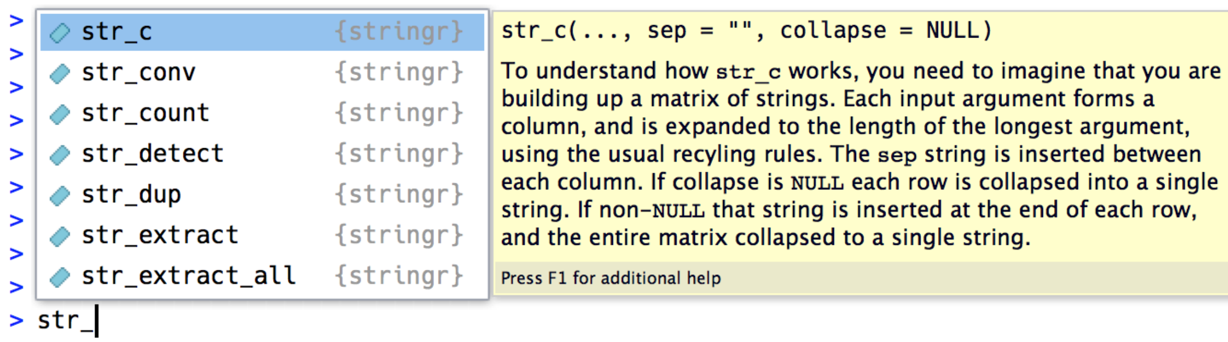
- **String Length**

Base R contains many functions to work with strings but we'll avoid them because they can be inconsistent, which makes them hard to remember. Instead, we'll use functions from stringr. These have more intuitive names, and all start with `str_`.

For example, `str_length()` tells you the number of characters in a string:

```
str_length(c("a", "R for data science", NA))
#> [1] 1 18 NA
```

The common `str_` prefix is particularly useful if you use RStudio,because typing `str_` will trigger autocomplete, allowing you to see all stringr functions:



- **Combining Strings**
  To combine two or more strings, use `str_c()` :

```
 str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Use the sep argument to control how they're separated:

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

Like most other functions in R, missing values are contagious. If you want them to print as "NA" , use `str_replace_na()` :

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

As shown in the preceding code, `str_c()` is vectorized, and it automatically recycles shorter vectors to the same length as the longest:

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Objects of length 0 are silently dropped. This is particularly useful in conjunction with if :

```
name <- "Hadley"
time_of_day <- "morning"
birthday <- FALSE
str_c(
"Good ", time_of_day, " ", name,
if (birthday) " and HAPPY BIRTHDAY",
"."
)
#> [1] "Good morning Hadley."
```

To collapse a vector of strings into a single string, use collapse :

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

Subsetting Strings
You can extract parts of a string using `str_sub()` . As well as the string, `str_sub()` takes start and end arguments that give the (inclusive) position of the substring:

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"
# negative numbers count backwards from end
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

Note that `str_sub()` won't fail if the string is too short; it will just return as much as possible:

```
str_sub("a", 1, 5)
#> [1] "a"
You can also use the assignment form of str_sub() to modify
strings:
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple" "banana" "pear"
```

- **Locales**
  Earlier I used `str_to_lower()` to change the text to lowercase. You can also use `str_to_upper()` or `str_to_title()` . However, changing case is more complicated than it might at first appear because different languages have different rules for changing case.You can pick which set of rules to use by specifying a locale:

```
# Turkish has two i's: with and without a dot, and it
# has different rules for capitalizing them:
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

The locale is specified as an ISO 639 language code, which is a two-or-three-letter abbreviation. If you don't already know the code for your language, Wikipedia has a good list. If you leave the locale blank, it will use the current locale, as provided by your operating system.

Another important operation that's affected by the locale is sorting. The base R `order()` and `sort()` functions sort strings using the current locale. If you want robust behavior across different computers, you may want to use `str_sort()` and `str_order()`, which take an additional locale argument:

```
x <- c("apple", "eggplant", "banana")

str_sort(x, locale = "en") # English
#> [1] "apple" "banana" "eggplant"

str_sort(x, locale = "haw") # Hawaiian
#> [1] "apple" "eggplant" "banana"
```

Missing Values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, a value can be missing in one of two possible ways:

- Explicitly, i.e., flagged with NA.
- Implicitly, i.e., simply not present in the data.

Let's illustrate this idea with a very simple dataset:

```
stocks <- tibble(
year= c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
qtr= c(1,2,3,4,2,3,4),
return = c(1.88, 0.59, 0.35,NA, 0.92, 0.17, 2.66)
)
```

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is explicitly missing, because the cell where its value should be instead contains NA .
- The return for the first quarter of 2016 is implicitly missing because it simply does not appear in the dataset.

One way to think about the difference is with this Zen-like koan: an explicit missing value is the presence of absence; an implicit missing value is the absence of a presence.

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

```
stocks %>%
   spread(year, return)
#> # A tibble: 4 × 3
#>
       Qtr  `2015``2016`
#> * <dbl> <dbl> <dbl>
#> 1    1      1.88    NA
#> 2   2       0.59    0.92
#> 3   3       0.35    0.17
#> 4   4       NA     2.66
```

Because these explicit missing values may not be important in other representations of the data, you can set `na.rm = TRUE` in `gather()` to turn explicit missing values implicit:

```
stocks %>%
    spread(year, return) %>%
    gather(year, return, `2015`:`2016`, na.rm = TRUE)
#> # A tibble: 6 × 3
#>      qtr     year    return
#> *   <dbl>  <chr>   <dbl>
#> 1    1      2015    1.88
#> 2     2     2015    0.59
#> 3    3      2015    0.35
#> 4    2      2016    0.92
#> 5    3      2016    0.17
#> 6    4      2016    2.66
```

Another important tool for making missing values explicit in tidy data is `complete()` :

```
stocks %>%
  complete(year, qtr)
#> # A tibble: 8 × 3
#>   year  qtr  return
#> <dbl> <dbl> <dbl>
#> 1 2015  1      1.88
#> 2 2015  2      0.59
#> 3 2015  3      0.35
#> 4 2015  4      NA
#> 5 2016  1      NA
#> 6 2016  2      0.92
```

```
#> # ... with 2 more rows
```

`complete()` takes a set of columns, and finds all unique combinations. It then ensures the original dataset contains all those values, filling in explicit NA s where necessary.

There's one other important tool that you should know for working with missing values. Sometimes when a data source has primarily been used for data entry, missing values indicate that the previous value should be carried forward:

```
treatment <- tribble(
 ~ person,          ~ treatment,     ~response,
"Derrick Whitmore",        1,            7,
NA,                        2,            10,
NA,                        3,            9,
"Katherine Burke",         1,             4
)
```

You can fill in these missing values with `fill()`. It takes a set of columns where you want missing values to be replaced by the most recent nonmissing value (sometimes called last observation carried forward):

```
treatment %>%
   fill(person)
#> # A tibble: 4 × 3
#>   person                 treatment        response
#>   <chr>                   <dbl>             <dbl>
#> 1 Derrick Whitmore          1                 7
#> 2 Derrick Whitmore          2                10
#> 3 Derrick Whitmore          3                 9
#> 4 Katherine Burke           1                 4
```

R Programming
Module 4: Working with dplyr in R

Learning objectives

1. Describe the purpose of the dplyr packages.
2. Use the split-apply-combine concept for data analysis.
3. Applying a filter, selecting specific columns.
4. Sorting data, adding or deleting columns and aggregating data.

Introduction

Visualization is an important tool for insight generation, but it is rare that you get the data in exactly the right form you need. Here we will learn how to transform your data using the dplyr package and illustrate the key ideas using data from the nycflights13 package, and use ggplot2 to help us understand the data.

```
library(nycflights13)
library(tidyverse)
```

dplyr

Here you are going to learn the five key dplyr functions that allow you to solve the vast majority of your data-manipulation challenges:

- Pick observations by their values (`filter()`)
- Reorder the rows (`arrange()`)
- Pick variables by their names (`select()`)
- Create new variables with functions of existing variables (`mutate()`)
- Collapse many values down to a single summary (`summarize()`)

These can all be used in conjunction with `group_by()`, which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

- The first argument is the data frame.
- The subsequent arguments describe what to do with the dataframe, using the variable names (without quotes).
- The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. Let's dive in and see how these verbs work.

Filter Rows with `filter()`

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
# A tibble: 842 x 19
      year  month    day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay
carrier
  <int> <int> <int>    <int>          <int>     <dbl>     <int>          <int>      <dbl> <chr>
 1  2013     1     1      517            515         2       830            819         11 UA
 2  2013     1     1      533            529         4       850            830         20 UA
 3  2013     1     1      542            540         2       923            850         33 AA
 4  2013     1     1      544            545        -1      1004           1022        -18 B6
 5  2013     1     1      554            600        -6       812            837        -25 DL
 6  2013     1     1      554            558        -4       740            728         12 UA
```

```
 7  2013     1     1      555             600        -5      913             854         19 B6
 8  2013     1     1      557             600        -3      709             723        -14 EV
 9  2013     1     1      557             600        -3      838             846         -8 B6
10  2013     1     1      558             600        -2      753             745          8 AA
# ... with 832 more rows, and 9 more variables.
```

When you run that line of code, **dplyr** executes the filtering operation and returns a new data frame. dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`:

```
jan1 <- filter(flights, month 1, day == 1)
```

R either prints out the results or saves them to a variable. If you want to do both, you can wrap the assignment in parentheses:

```
(dec25 <- filter(flights, month==12, day==25))
# A tibble: 719 x 19
      year  month    day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay
carrier
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
 1  2013    12    25      456             500        -4      649             651         -2 US
 2  2013    12    25      524             515         9      805             814         -9 UA
 3  2013    12    25      542             540         2      832             850        -18 AA
 4  2013    12    25      546             550        -4     1022            1027         -5 B6
 5  2013    12    25      556             600        -4      730             745        -15 AA
 6  2013    12    25      557             600        -3      743             752         -9 DL
 7  2013    12    25      557             600        -3      818             831        -13 DL
 8  2013    12    25      559             600        -1      855             856         -1 B6
 9  2013    12    25      559             600        -1      849             855         -6 B6
10  2013    12    25      600             600         0      850             846          4 B6
# ... with 709 more rows, and 9 more variables.
```

Comparisons

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `<`, `<=`, `!=(not equal)`, and `==` `(equal)`. When you're starting out with R, the easiest mistake to make is to use = instead of == when testing for equality. When this happens you'll get an informative error:

```
filter(flights, month = 1)
# Error: filter() takes unnamed arguments. Do you need '=='?
```

There's another common problem you might encounter when using **==** floating-point numbers. These results might surprise you!

```
> sqrt(2) ^ 2 == 2
[1] FALSE
> 1/49*49 == 1
[1] FALSE
```

Computers use finite precision arithmetic (they obviously can't store an infinite number of digits!) so remember that every number you see is an approximation. Instead of relying on ==, use `near()`:

```
> near(sqrt(2) ^ 2, 2)
[1] TRUE
> near(1 / 49, 1)
[1] FALSE
```

Logical Operators

Multiple arguments to `filter()` are combined with "and": every expression must be true in order for a row to be included in the output. For other types of combinations, you'll need to use Boolean operators yourself: `&` is "and," `|` is "or," and ! is "not." The following figure shows the complete set of Boolean operations. The following code finds all flights that departed in November or December:

```
filter(flights, month == 1 | month == 2)
```

The order of operations doesn't work like English. You can't write `filter(Flights, month == 11 | 12)`, which you might literally translate into "finds all flights that departed in November or December. Instead, it finds all months that equal 11 `|` 12, an expression that evaluates to TRUE. In a numeric context (like here), TRUE becomes one, so this finds all flights in January, not November or December. This is quite confusing!

A useful shorthand for this problem is `x %in% y`. This will select every row where `x` is one of the values in `y`. We could use it to rewrite the preceding code:

```
Nov_dec <- filter(flights, month %in% c(11, 12))
```

Sometimes you can simplify complicated subsetting by remembering De Morgan's law: `!(x & y)` is the same as `!x | !y`, and `!(x | y)` is the same as `!x & !y`. For example, if you wanted to find flights that weren't delayed (on arrival or departure) by more than two hours, you could use either of the following two filters:

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <=120))
```

As well as `&` and `|`, R also has `&&` and `||`. Whenever you start using complicated, multipart expressions in `filter()`, consider making them explicit variables instead. That makes it much easier to check your work. You'll learn how to create new variables shortly.

Missing Values

One important feature of R that can make comparison tricky is missing values or NAs ("not available"). NA represents an unknown value so missing values are "contagious"; almost any operation involving an unknown value will also be unknown:

```
NA > 5
[1] NA
10 == NA
[1] NA
NA + 10
```

```
[1] NA
NA / 2
[1] NA
```

The most confusing result is this one:

```
> NA == NA
[1] NA
```

It's easiest to explain why this is true with a bit more context:

```
# Let x be Mary's age. We don't know how old she is.
x <- NA

# Let y be John's age. We don't know how old he is.
y <- NA

# Are John and Mary the same age?
x == y
[1] NA
# We don't know!
```

If you want to determine if a value is missing, use `is.na()`:

```
is.na(x)
[1] TRUE
```

`filter()` only includes rows where the condition is `TRUE;` it excludes both FALSE and NA values. If you want to preserve missing values, ask for them explicitly:

```
> df <- tibble(x = c(1, NA, 3))
> filter(df, x > 1)
# A tibble: 1 x 1
      x
  <dbl>
1     3

> filter(df, is.na(x) | x > 1)
# A tibble: 2 x 1
      x
  <dbl>
1    NA
2     3
```

Arrange Rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
> arrange(flights, year, month, day)
# A tibble: 336,776 x 19
      year  month    day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay
carrier
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
 1  2013     1     1      517            515         2      830            819        11 UA
 2  2013     1     1      533            529         4      850            830        20 UA
 3  2013     1     1      542            540         2      923            850        33 AA
 4  2013     1     1      544            545        -1     1004           1022       -18 B6
 5  2013     1     1      554            600        -6      812            837       -25 DL
 6  2013     1     1      554            558        -4      740            728        12 UA
 7  2013     1     1      555            600        -5      913            854        19 B6
 8  2013     1     1      557            600        -3      709            723       -14 EV
 9  2013     1     1      557            600        -3      838            846        -8 B6
10  2013     1     1      558            600        -2      753            745         8 AA
# ... with 336,766 more rows, and 9 more variables.
```

Use `desc()` to reorder by a column in descending order:

```
> arrange(flights, desc(arr_delay))
# A tibble: 336,776 x 19
      year  month    day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay
carrier
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
 1  2013     1     9      641            900      1301     1242           1530      1272 HA
 2  2013     6    15     1432           1935      1137     1607           2120      1127 MQ
 3  2013     1    10     1121           1635      1126     1239           1810      1109 MQ
 4  2013     9    20     1139           1845      1014     1457           2210      1007 AA
 5  2013     7    22      845           1600      1005     1044           1815       989 MQ
 6  2013     4    10     1100           1900       960     1342           2211       931 DL
 7  2013     3    17     2321            810       911      135           1020       915 DL
 8  2013     7    22     2257            759       898      121           1026       895 DL
 9  2013    12     5      756           1700       896     1058           2020       878 AA
10  2013     5     3     1133           2055       878     1250           2215       875 MQ
# ... with 336,766 more rows, and 9 more variables.
```

Missing values are always sorted at the end:

```
> df <- tibble(x = c(5, 2, NA))
> arrange(df, x)
# A tibble: 3 x 1
      x
  <dbl>
1     2
2     5
3    NA

> arrange(df, desc(x))
# A tibble: 3 x 1
      x
  <dbl>
1     5
2     2
3    NA
```

Select Columns with `select()`

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

`select()` is not terribly useful with the flight data because we only have 19 variables, but you can still get a general idea:

```
# Select columns by name
> select(flights, year, month, day)

# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 # ... with 336,768 more rows
```

```
# Select all columns between year and day (inclusive)
> select(flights, year:day)

# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# ... with 336,766 more rows
```

```
# Select all columns except those from year to day (inclusive)
> select(flights, -(year:day))

# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier flight tailnum
      <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>    <int> <chr>
 1      517            515         2      830            819        11 UA        1545 N14228
 2      533            529         4      850            830        20 UA        1714 N24211
 3      542            540         2      923            850        33 AA        1141 N619AA
 4      544            545        -1     1004           1022       -18 B6         725 N804JB
 5      554            600        -6      812            837       -25 DL         461 N668DN
```

```
 6      554             558       -4      740             728         12 UA         1696 N39463
 7      555             600       -5      913             854         19 B6          507 N516JB
 8      557             600       -3      709             723        -14 EV         5708 N829AS
 9      557             600       -3      838             846         -8 B6           79 N593JB
10      558             600       -2      753             745          8 AA          301 N3ALAA
# ... with 336,766 more rows, and 7 more variables.
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")` matches names that begin with "ab"'.
- `ends_with("xyz")` matches names that end with "xyz".
- `contains("ijk")` matches names that contain "ijk".
- `matches("(.)\\1")` selects variables that match a regular expression. This one matches any variables that contain repeated characters.
- `num_range("x", 1:3)` matches x1, x2, and x3.

`select()` can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use `rename()`, which is a variant of select that keeps all the variables that aren't explicitly mentioned:

```
> rename(flights, tail_num=tailnum)
# A tibble: 336,776 x 19
      year  month    day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay
carrier
 <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
 1  2013     1     1      517            515         2      830            819        11 UA
 2  2013     1     1      533            529         4      850            830        20 UA
 3  2013     1     1      542            540         2      923            850        33 AA
 4  2013     1     1      544            545        -1     1004           1022       -18 B6
 5  2013     1     1      554            600        -6      812            837       -25 DL
 6  2013     1     1      554            558        -4      740            728        12 UA
 7  2013     1     1      555            600        -5      913            854        19 B6
 8  2013     1     1      557            600        -3      709            723       -14 EV
 9  2013     1     1      557            600        -3      838            846        -8 B6
10  2013     1     1      558            600        -2      753            745         8 AA
# ... with 336,766 more rows, and 9 more variables.
```

Another option is to use `select()` in conjunction with `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame:

```
> select(flights, time_hour, air_time, everything())
# A tibble: 336,776 x 19

 time_hour           air_time  year month   day dep_time sched_dep_time dep_delay arr_time
   <dttm>                 <dbl> <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1 2013-01-01 05:00:00      227  2013     1     1      517            515         2      830
 2 2013-01-01 05:00:00      227  2013     1     1      533            529         4      850
 3 2013-01-01 05:00:00      160  2013     1     1      542            540         2      923
 4 2013-01-01 05:00:00      183  2013     1     1      544            545        -1     1004
 5 2013-01-01 06:00:00      116  2013     1     1      554            600        -6      812
 6 2013-01-01 05:00:00      150  2013     1     1      554            558        -4      740
 7 2013-01-01 06:00:00      158  2013     1     1      555            600        -5      913
 8 2013-01-01 06:00:00       53  2013     1     1      557            600        -3      709
```

```
 9 2013-01-01 06:00:00      140  2013     1     1      557            600         -3       838
10 2013-01-01 06:00:00      138  2013     1     1      558            600         -2       753
# ... with 336,766 more rows, and 10 more variables.
```

Mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables. Remember that when you're in RStudio, the easiest way to see all the columns is `View()`:

```
> flights_sml <- select(flights,
      Year:day,
      ends_with("delay"),
      distance, air_time
)
> mutate(flights_sml,
      gain = arr_delay - dep_delay,
      speed = distance / air_time * 100
)
# A tibble: 336,776 x 9
    year month   day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9  617.
 2  2013     1     1         4        20     1416      227    16  624.
 3  2013     1     1         2        33     1089      160    31  681.
 4  2013     1     1        -1       -18     1576      183   -17  861.
 5  2013     1     1        -6       -25      762      116   -19  657.
 6  2013     1     1        -4        12      719      150    16  479.
 7  2013     1     1        -5        19     1065      158    24  674.
 8  2013     1     1        -3       -14      229       53   -11  432.
 9  2013     1     1        -3        -8      944      140    -5  674.
10  2013     1     1        -2         8      733      138    10  531.
# ... with 336,766 more rows.
```

Note that you can refer to columns that you've just created:

```
> mutate(flights_sml,
      gain = arr_delay - dep_delay,
      hours = air_time / 60,
      gain_per_hour = gain / hours
)
# A tibble: 336,776 x 10
    year month   day dep_delay arr_delay distance air_time  gain hours gain_per_hour
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>         <dbl>
 1  2013     1     1         2        11     1400      227     9 3.78           2.38
 2  2013     1     1         4        20     1416      227    16 3.78           4.23
 3  2013     1     1         2        33     1089      160    31 2.67          11.6
 4  2013     1     1        -1       -18     1576      183   -17 3.05          -5.57
```

```
 5  2013     1     1         -6       -25       762       116   -19 1.93           -9.83
 6  2013     1     1         -4        12       719       150    16 2.5             6.4
 7  2013     1     1         -5        19      1065       158    24 2.63            9.11
 8  2013     1     1         -3       -14       229        53   -11 0.883         -12.5
 9  2013     1     1         -3        -8       944       140    -5 2.33           -2.14
10  2013     1     1         -2         8       733       138    10 2.3             4.35
# ... with 336,766 more rows
```

If you only want to keep the new variables, use `transmute()`:

```
> transmute(flights,
      gain = arr_delay - dep_delay,
      hours = air_time / 60,
      gain_per_hour = gain / hours
)
```

```
# A tibble: 336,776 x 3
    gain hours gain_per_hour
   <dbl> <dbl>         <dbl>
 1     9 3.78           2.38
 2    16 3.78           4.23
 3    31 2.67          11.6
 4   -17 3.05          -5.57
 5   -19 1.93          -9.83
 6    16 2.5            6.4
 7    24 2.63           9.11
 8   -11 0.883        -12.5
 9    -5 2.33          -2.14
10    10 2.3            4.35
# ... with 336,766 more rows
```

Useful Creation Functions

There are many functions for creating new variables that you use with mutate(). The key property is that the function must be vectorized: it must take a vector of values as input, and returns a vector with the same number of values as output. There's no way to list every possible function that you might use, but here's a selection of functions that are frequently useful:

*Arithmetic operators +, -, *, /, ^*

These are all vectorized, using the so-called "recycling rules". If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: `air_time / 60, hours * 6 + minute`, etc.

Arithmetic operators are also useful in conjunction with the aggregate functions you'll learn about later. For example, `x / sum(x)` calculates the proportion of a total, and `y - mean(y)` computes the difference from the mean.

Modular arithmetic (%/% and %%)

%/% (integer division) and %% (remainder), where x == y * (x%/% y) + (x %% y). Modular arithmetic is a handy tool because it allows you to break integers into pieces. For example, in the flights dataset, you can compute hour and minute from `dep_time` with:

```
> transmute(flights, dep_time, hour = dep_time %/% 100, minute
= dep_time %% 100)

# A tibble: 336,776 x 3
   dep_time  hour minute
      <int> <dbl>  <dbl>
 1      517     5     17
 2      533     5     33
 3      542     5     42
 4      544     5     44
 5      554     5     54
 6      554     5     54
 7      555     5     55
 8      557     5     57
 9      557     5     57
10      558     5     58
# ... with 336,766 more rows
```

Logs `log(), log2(), log10()`

Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude. They also convert multiplicative relationships to the additive.

All else being equal, I recommend using log2 because it's easy to interpret: a difference of on the log scale corresponds to doubling on the original scale and a difference of -1 corresponds to halving.

Offsets

`lead()` and `lag()` allow you to refer to leading or lagging values. This allows you to compute running differences (e.g, `x - lag(x)`) or find when values change (`x != lag (x)`). They are most useful in conjunction with `group_by()`:

```
> (x <- 1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> lag(x)
 [1] NA  1  2  3  4  5  6  7  8  9
> lead(x)
 [1]  2  3  4  5  6  7  8  9 10 NA
```

Cumulative and rolling aggregates

R provides functions for running sums, products, mins, and maxes: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; and **dplyr** provides `cummean()` for cumulative means. If you need rolling aggregates (i.e., a sum computed over a rolling window), try the **RcppRoll** package:

```
  x
 [1]  1  2  3  4  5  6  7  8  9 10
> cumsum(x)
 [1]  1  3  6 10 15 21 28 36 45 55
> cummean(x)
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

Logical comparisons <, <=, >, >=, !=

If you're doing a complex sequence of logical operations it's often a good idea to store the interim values in new variables so you can check that each step is working as expected.

Ranking

There are a number of ranking functions, but you should start with `min_rank()`. It does the most usual type of ranking (e.g. first, second, third, fourth). The default gives the smallest values the smallest ranks; use `desc(x)` to give the largest value smallest ranks:

```
y <- c(1, 2, 2, NA, 3, 4)
> min_rank(y)
[1]  1  2  2 NA  4  5
> min_rank(desc(y))
[1]  5  3  3 NA  2  1
```

If `min_rank()` doesn't do what you need, look at the variants `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()` and `ntile()`. See their help pages for more details:

```
> row_number(y)
[1]  1  2  3 NA  4  5
> dense_rank(y)
[1]  1  2  2 NA  3  4
> percent_rank(y)
[1] 0.00 0.25 0.25   NA 0.75 1.00
> cume_dist(y)
[1] 0.2 0.6 0.6  NA 0.8 1.0
```

Group Summaries with summarize()

The last key verb is `summarize()`. It collapses a data frame to a single row:

```
> summarize(flights, delay = mean(dep_delay, na.rm = TRUE))
# A tibble: 1 x 1
  delay
  <dbl>
1 12.6
```

`summarize()` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the **dplyr** verbs on a grouped data frame they'll be automatically applied "by group." For example, if we applied exactly the same code to a data frame grouped by date, we get the average delay per date:

```
> by_day <- group_by(flights, year, month, day)
> summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

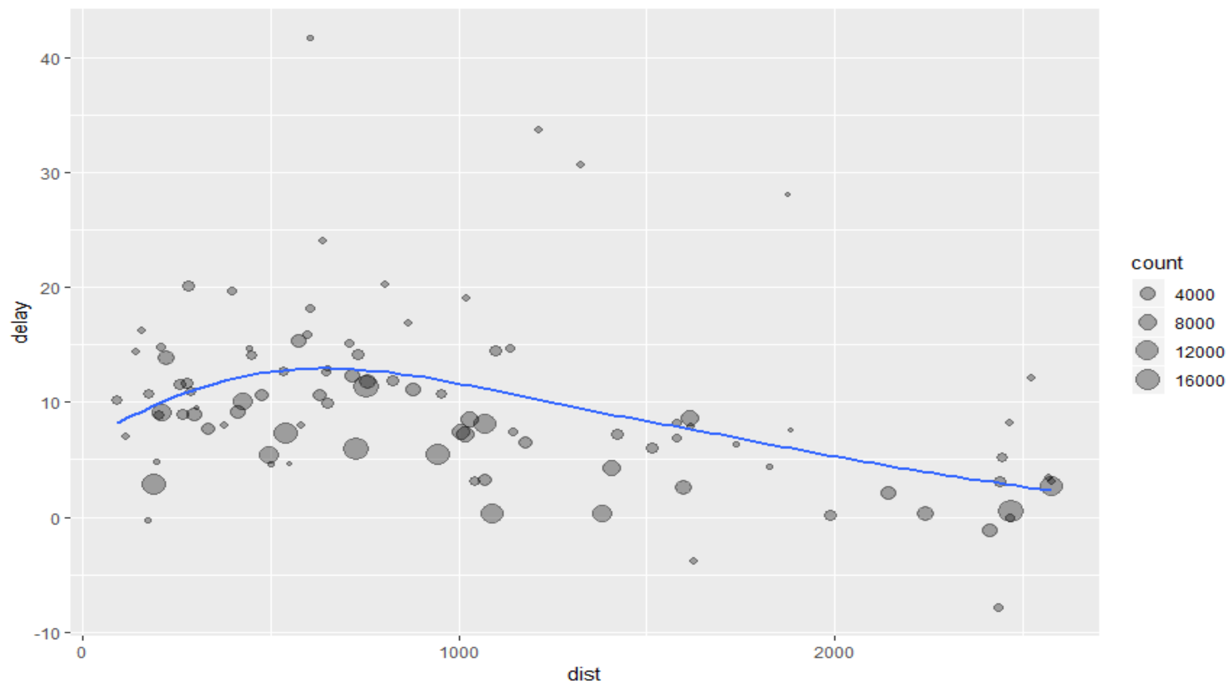
```
# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

Together `group_by()` and `summarize()` provide one of the tools that you'll use most commonly when working with **dplyr**: grouped summaries. But before we go any further with this, we need to introduce a powerful new idea: the pipe.

Combining Multiple Operations with Pipe

Imagine that we want to explore the relationship between the distance and average delay for each location. Using what you know about dplyr, you might write code like this:

```
by_dest <- group_by(flights, dest)
delay <- summarize(by_dest, count = n(), dist = mean(distance, na.rm
= TRUE), delay = mean(arr_delay, na.rm = TRUE))
delay <- filter(delay, count > 20, dest != "HNL")

# It looks like delays increase with distance up to -750 miles
# and then decrease. Maybe as flights get longer there's more
# ability to make up delays in the air?

ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```



There are three steps to prepare this data:

1. Group flights by destination.
2. Summarize to compute distance, average delay, and number of flights.
3. Filter to remove noisy points and Honolulu airport, whicalmost twice as far away as the next closest airport.

This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard, so this slows down our analysis.

There's another way to tackle the same problem with the pipe, `%>%`:

```
delays <- flights %>%
  group_by(dest) %>%
  summarize(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

This focuses on the transformations, not what's being transformed which makes the code easier to read. You can read it as a series of imperative statements: group, then summarize, then filter. As suggested by this reading, a good way to pronounce `%>%` when reading code is "then."

Behind the scenes, `X %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z )` turns into `g(f(x, y), z)`, and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom. We'll use piping frequently from now on because it considerably improves the readability of code.

Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is ggplot2: it was written before the pipe was discovered. Unfortunately, the next iteration of ggplot2, ggvis, which does use the pipe, isn't ready for prime time yet.

Missing Values

You may have wondered about the `na.rm` argument we used earlier. What happens if we don't set it?

```
> flights %>%
      group_by(year, month, day) %>%
      summarize(mean = mean(dep_delay))
```

```
# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# ... with 355 more rows
```

We get a lot of missing values! That's because aggregation functions obey the usual rule of missing values: if there's any missing values in the input, the output will be a missing value. Fortunately, all aggregation functions have an `na.rm` argument, which removes the missing values prior to computation:

```
> flights %>%
      group_by(year, month, day) %>%
      summarize(mean = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
```

```
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

In this case, where missing values represent cancelled flights, we could also tackle the problem by first removing the cancelled flights.We'll save this dataset so we can reuse it in the next few examples:

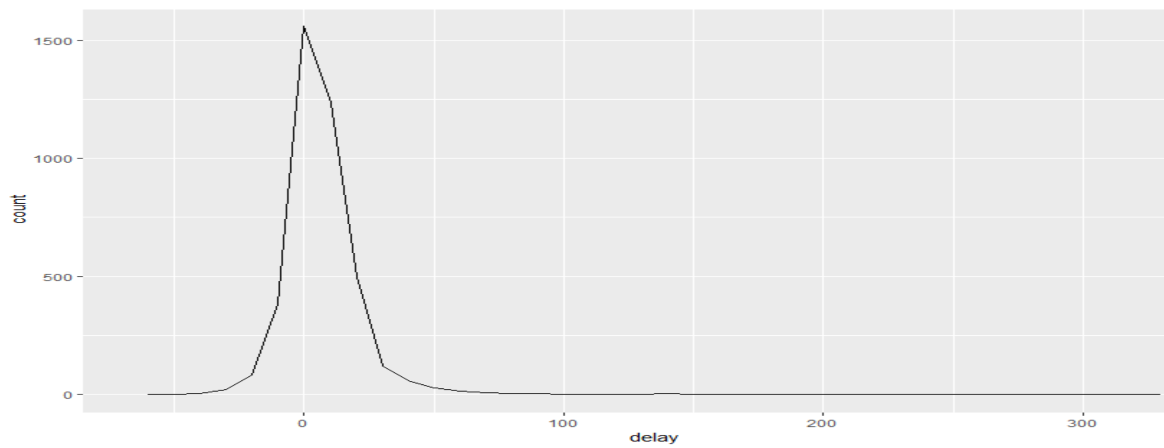
```
> not_cancelled <-  flights %>%
      filter(!is.na(dep_delay), !is.na(arr_delay))

> not_cancelled %>%
      group_by(year, month, day) %>%
      summarize(mean = mean(dep_delay))
```

```
# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.4
 2  2013     1     2 13.7
 3  2013     1     3 10.9
 4  2013     1     4  8.97
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.56
 9  2013     1     9  2.30
10  2013     1    10  2.84
# ... with 355 more rows
```

Counts

Whenever you do any aggregation, it's always a good idea to include either a count `(n())`, or count of nonmissing missing value (`sum(!is.na(x))`). That way you can check that you're not drawing conclusions based on very small amounts of data. For example, let's look at the planes (identified by their tail number) that have the highest average delays:

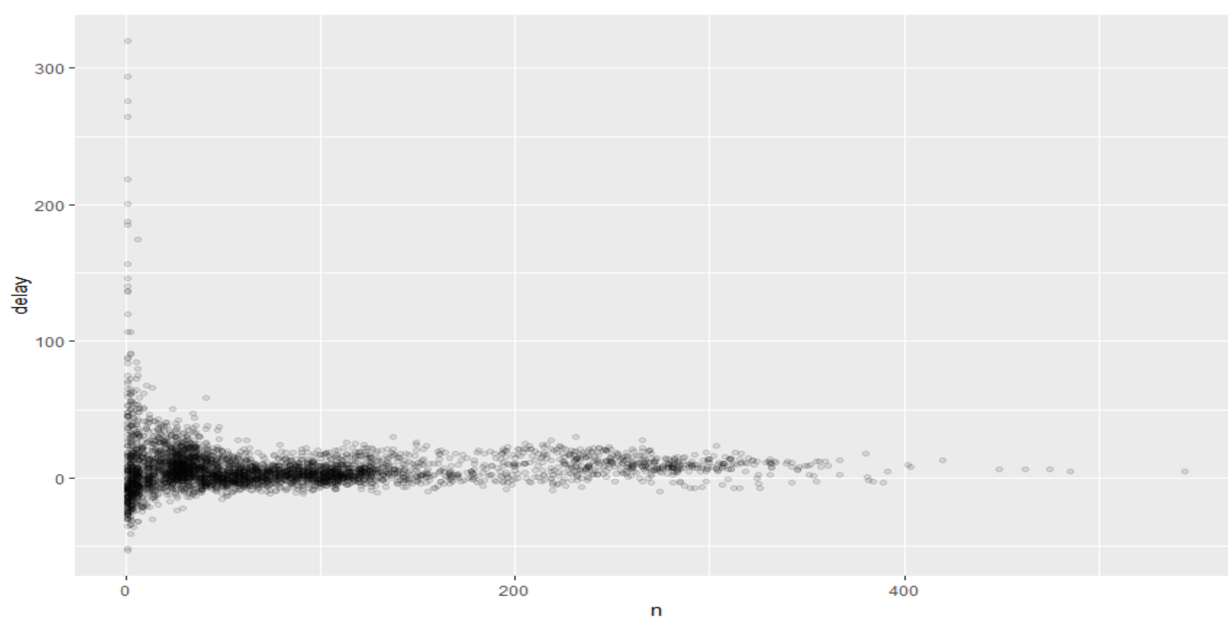
```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay)
  )
ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```



Wow, there are some planes that have an *average* delay of 5 hours (300 minutes)!

The story is actually a little more nuanced. We can get more insight if we draw a scatterplot of number of flights versus average delay:

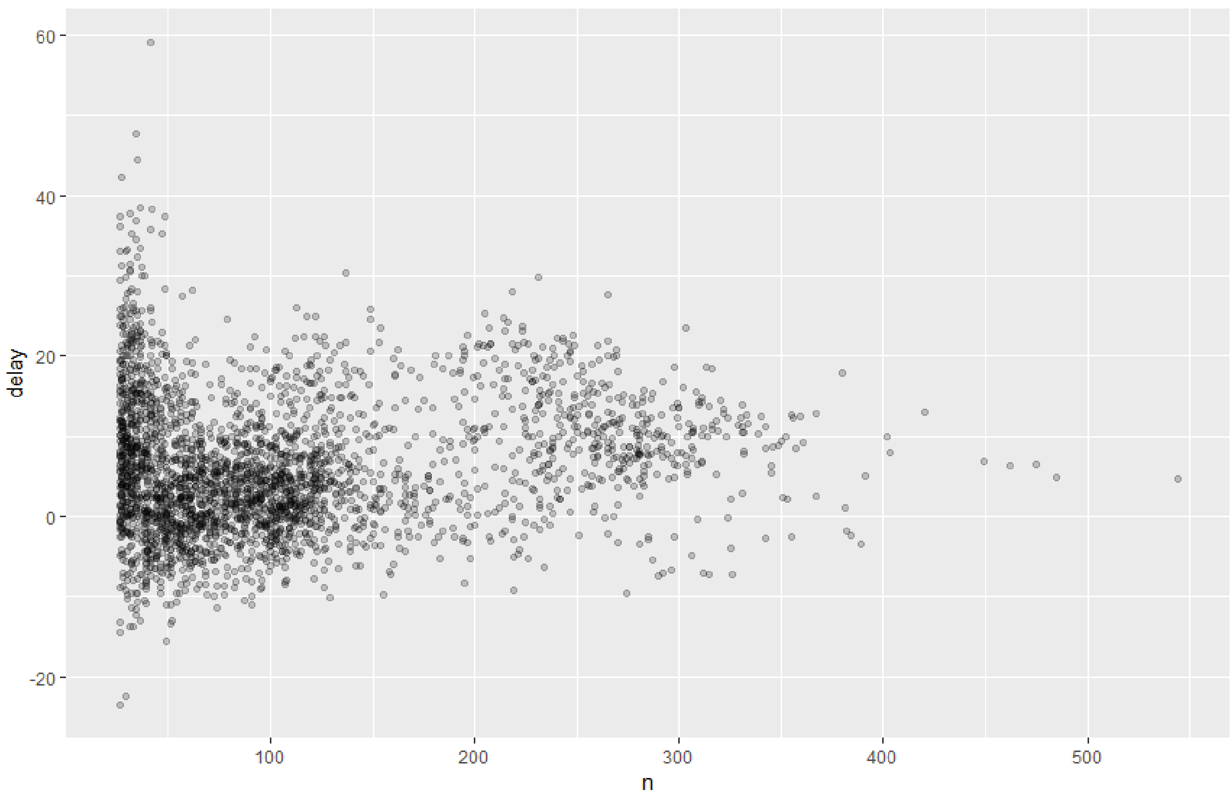
```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



Not surprisingly, there is much greater variation in the average delay when there are few flights. The shape of this plot is very characteristic: whenever you plot a mean (or other summary) versus group size, you'll see that the variation decreases as the sample size increases.

When looking at this sort of plot, is often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups. This is what the following code does, as well as showing you a handy pattern for integrating **ggplot2** into **dplyr** flows. It's a bit painful that you have to switch from `%>%` to +, but once you get the hang of it, it's quite convenient:

```
delays %>%
  filter(n >25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



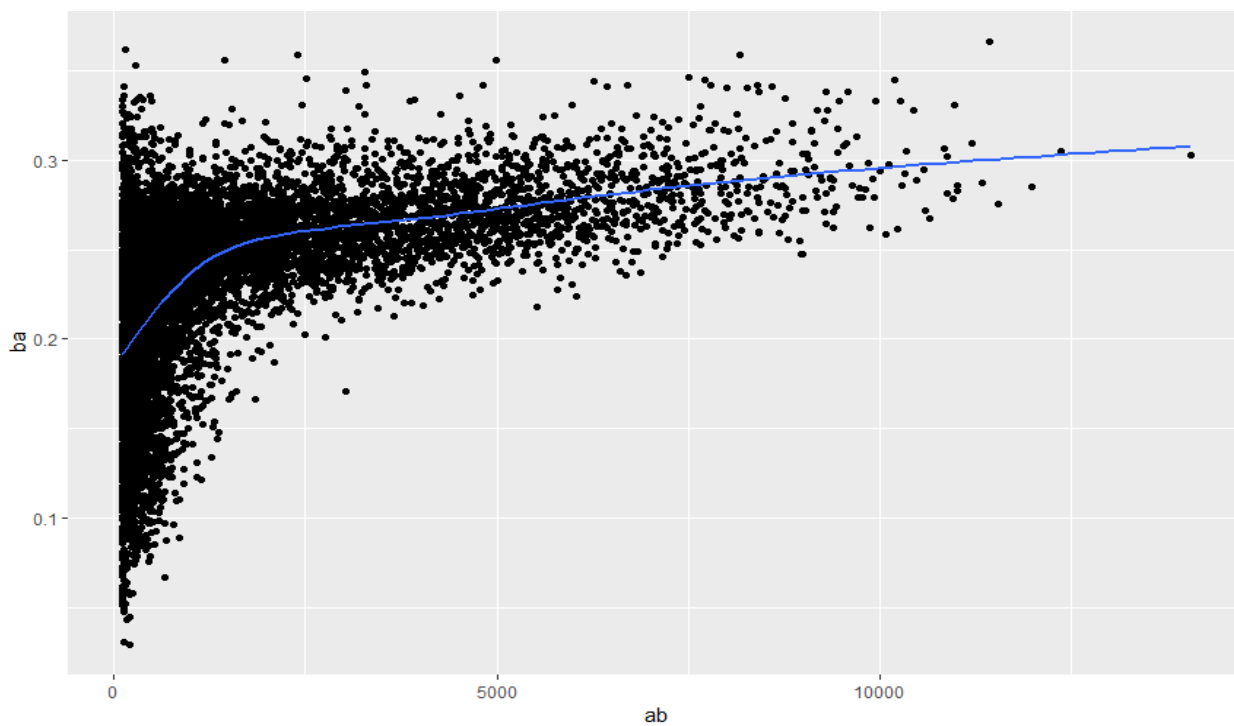
There's another common variation of this type of pattern. Let's look at how the average performance of batters in baseball is related to the number of times they're at bat. Here I use data from the **Lahman** package to compute the batting average (number of hits / number of attempts) of every major league baseball player.

When I plot the skill of the batter (measured by the batting average, `ba`) against the number of opportunities to hit the ball (measured by at bat, `ab`), you see two patterns:

- As above, the variation in our aggregate decreases as we get more data points.

- There's a positive correlation between skill (`ba`) and opportunities to hit the ball (`ab`). This is because teams control who gets to play, and obviously they'll pick their best players:

```
# Convert to a tibble so it prints nicely
batting <- as_tibble(Lahman::Batting)

batters <- batting %>%
  group_by(playerID) %>%
  summarize(
    ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
    ab = sum(AB, na.rm = TRUE)
)

batters %>%
  filter(ab > 100) %>%
  ggplot(mapping = aes(x = ab, y = ba)) +
  geom_point() +
  geom_smooth(se = FALSE)
```



This also has important implications for ranking. If you naively sort on `desc(ba)`, the people with the best batting averages are clearly lucky, not skilled:

```
> batters %>%
     arrange(desc(ba))
```

```
# A tibble: 18,915 x 3
   playerID     ba    ab
   <chr>     <dbl> <int>
 1 abramge01     1     1
 2 banisje01     1     1
 3 bartocl01     1     1
 4 bassdo01      1     1
 5 berrijo01     1     1
 6 birasst01     1     2
 7 bruneju01     1     1
 8 burnscb01     1     1
 9 cammaer01     1     1
10 campsh01      1     1
# ... with 18,905 more rows
```

Useful Summary Functions

Just using means, counts, and sum can get you a long way, but R provides many other useful summary functions:

Measures of location

We've used `mean(x)`, but `median(x)` is also useful. The mean is the sum divided by the length: the median is a value where 50%of x is above it, and 50% is below it.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    # average delay:
    avg_delay1 = mean(arr_delay),
    # average positive delay:
    avg_delay2 = mean(arr_delay[arr_delay > 0])
  )
```

```
# A tibble: 365 x 5
# Groups:   year, month [?]
    year month   day avg_delay1 avg_delay2
   <int> <int> <int>      <dbl>      <dbl>
 1  2013     1     1     12.7         32.5
 2  2013     1     2     12.7         32.0
 3  2013     1     3      5.73        27.7
 4  2013     1     4     -1.93        28.3
 5  2013     1     5     -1.53        22.6
 6  2013     1     6      4.24        24.4
 7  2013     1     7     -4.95        27.8
 8  2013     1     8     -3.23        20.8
 9  2013     1     9     -0.264       25.6
10  2013     1    10     -5.90        27.3
# ... with 355 more rows
```

Measures of spread `sd(x), IQR(x), mad(x)`

The mean squared deviation, or standard deviation or sd for short, is the standard measure of spread. The interquartile range `IQR()` and median absolute deviation `mad(x)` are robust equivalents that may be more useful if you have outliers:

```
# Why is the distance to some destinations variable is more
# than to others?
not_cancelled %>%
  group_by(dest) %>%
  summarize(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))

# A tibble: 104 x 2
   dest  distance_sd
   <chr>       <dbl>
 1 EGE         10.5
 2 SAN         10.4
 3 SFO         10.2
 4 HNL         10.0
 5 SEA          9.98
 6 LAS          9.91
 7 PDX          9.87
 8 PHX          9.86
 9 LAX          9.66
10 IND          9.46
# ... with 94 more rows
```

Measures of position `first(x), nth(x, 2), last(x)`

These work similarly to `x[1], x[2]`, and `x[length(x)]` but let you set a default value if that position does not exist (1e., you're trying to get the third element from a group that only has two elements).

For example, we can find the first and last departure for each day:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )

# A tibble: 365 x 5
# Groups:   year, month [?]
    year month   day first_dep last_dep
   <int> <int> <int>     <int>    <int>
 1  2013     1     1       517     2356
 2  2013     1     2        42     2354
 3  2013     1     3        32     2349
 4  2013     1     4        25     2358
 5  2013     1     5        14     2357
```

```
 6  2013     1     6        16     2355
 7  2013     1     7        49     2359
 8  2013     1     8       454     2351
 9  2013     1     9         2     2252
10  2013     1    10         3     2320
# ... with 355 more rows
```

These functions are complementary to filtering on ranks. Filtering gives you all variables, with each observation in a separate row:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  mutate(r = min_rank(desc(dep_time))) %>%
  filter(r %in% range(r))
```

```
# A tibble: 770 x 20
# Groups:   year, month, day [365]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1     2356           2359        -3      425            437
 3  2013     1     2       42           2359        43      518            442
 4  2013     1     2     2354           2359        -5      413            437
 5  2013     1     3       32           2359        33      504            442
 6  2013     1     3     2349           2359       -10      434            445
 7  2013     1     4       25           2359        26      505            442
 8  2013     1     4     2358           2359        -1      429            437
 9  2013     1     4     2358           2359        -1      436            445
10  2013     1     5       14           2359        15      503            445
# ... with 760 more rows, and 12 more variables.
```

Counts

You've seen `n()`, which takes no arguments, and returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct(unique) values, use `n_distinct(x)`:

```
# Which destinations have the most carriers?
not_cancelled %>%
  group_by(dest) %>%
  summarize(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
```

```
# A tibble: 104 x 2
   dest  carriers
   <chr>    <int>
 1 ATL          7
 2 BOS          7
 3 CLT          7
 4 ORD          7
 5 TPA          7
 6 AUS          6
 7 DCA          6
```

```
 8 DTW          6
 9 IAD          6
10 MSP          6
# ... with 94 more rows
```

Counts are so useful that **dplyr** provides a simple helper if all you want is a count

```
not_cancelled %>% count(dest)
```

```
# A tibble: 104 x 2
   dest      n
   <chr> <int>
 1 ABQ     254
 2 ACK     264
 3 ALB     418
 4 ANC       8
 5 ATL   16837
 6 AUS    2411
 7 AVL     261
 8 BDL     412
 9 BGR     358
# ... with 95 more rows
```

You can optionally provide a weight variable. For example, you could use this to "count" (sum) the total number of miles a plane flew:

```
> not_cancelled %>%
      count(tailnum, wt = distance)
```

```
# A tibble: 4,037 x 2
   tailnum      n
   <chr>    <dbl>
 1 D942DN    3418
 2 N0EGMQ  239143
 3 N10156  109664
 4 N102UW   25722
 5 N103US   24619
 6 N104UW   24616
 7 N10575  139903
 8 N105UW   23618
 9 N107US   21677
10 N108UW   32070
# ... with 4,027 more rows
```

`Counts and proportions of logical values sum(x> 10), mean(y == 0)`

When used with numeric functions, TRUE is converted to 1 and FALSE to 0. This makes `sum()` and `mean()` very useful: `sum(x)` gives the number of TRUEs in `x`, and `mean(x)` gives the proportion:

```
# How many flights left before 5am? (these usually
```

```
# indicate delayed flights from the previous day)
> not_cancelled %>%
group_by(year, month, day) %>%
summarize(n_early = sum(dep_time < 500))

# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day n_early
   <int> <int> <int>   <int>
 1  2013     1     1       0
 2  2013     1     2       3
 3  2013     1     3       4
 4  2013     1     4       3
 5  2013     1     5       3
 6  2013     1     6       2
 7  2013     1     7       2
 8  2013     1     8       1
 9  2013     1     9       3
10  2013     1    10       3
# ... with 355 more rows

# What proportion of flights are delayed by more
# than an hour?
> not_cancelled %>%
     group_by(year, month, day) %>%
     summarize(hour_perc = mean(arr_delay > 60))

# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day hour_perc
   <int> <int> <int>     <dbl>
 1  2013     1     1    0.0722
 2  2013     1     2    0.0851
 3  2013     1     3    0.0567
 4  2013     1     4    0.0396
 5  2013     1     5    0.0349
 6  2013     1     6    0.0470
 7  2013     1     7    0.0333
 8  2013     1     8    0.0213
 9  2013     1     9    0.0202
10  2013     1    10    0.0183
# ... with 355 more rows
```

Grouping by Multiple Variables

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll up a dataset:

```
daily <- group_by(flights, year, month, day)
(per_day <- summarize(daily, flights = n()))

# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day flights
```

```
   <int> <int> <int>   <int>
 1  2013     1     1     842
 2  2013     1     2     943
 3  2013     1     3     914
 4  2013     1     4     915
 5  2013     1     5     720
 6  2013     1     6     832
 7  2013     1     7     933
 8  2013     1     8     899
 9  2013     1     9     902
10  2013     1    10     932
# ... with 355 more rows
```

```
(per_month <- summarize(per_day, flights = sum(flights)))
```

```
# A tibble: 12 x 3
# Groups:   year [?]
    year month flights
   <int> <int>   <int>
 1  2013     1   27004
 2  2013     2   24951
 3  2013     3   28834
 4  2013     4   28330
 5  2013     5   28796
 6  2013     6   28243
 7  2013     7   29425
 8  2013     8   29327
 9  2013     9   27574
10  2013    10   28889
11  2013    11   27268
12  2013    12   28135
```

```
(per_year <- summarize(per_month, flights = sum(flights)))
```

```
# A tibble: 1 x 2
   year flights
  <int>   <int>
1  2013  336776
```

Be careful when progressively rolling up summaries: it's OK for sums and counts, but you need to think about weighting means and variances, and it's not possible to do it exactly for rank-based statistics like the median. In other words, the sum of groupwise sums is the overall sum, but the median of groupwise medians is not the overall median.

Ungrouping

If you need to remove grouping, and return to operations on ungrouped data, use `ungroup()`:

```
> daily %>%
      ungroup() %>%
      summarize(flights = n())
```

```
# A tibble: 1 x 1
  flights
```

```
    <int>
1  336776
```

Grouped Mutates (and Filters)

Grouping is most useful in conjunction with `summarize()`, but you can also do convenient operations with `mutate()` and `filter()`:

- Find the worst members of each group

```
flights_sml %>%
     group_by(year, month, day) %>%
     filter(rank(desc(arr_delay)) < 10)
```

```
# A tibble: 3,306 x 7
# Groups:   year, month, day [365]
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1       853       851      184       41
 2  2013     1     1       290       338     1134      213
 3  2013     1     1       260       263      266       46
 4  2013     1     1       157       174      213       60
 5  2013     1     1       216       222      708      121
 6  2013     1     1       255       250      589      115
 7  2013     1     1       285       246     1085      146
 8  2013     1     1       192       191      199       44
 9  2013     1     1       379       456     1092      222
10  2013     1     2       224       207      550       94
# ... with 3,296 more rows
```

- Find all groups bigger than a threshold:

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
```

```
# A tibble: 332,577 x 19
# Groups:   dest [77]
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 332,567 more rows, and 12 more variables.
```

- Standardize to compute per group metrics:

```
popular_dests %>%
      filter(arr_delay > 0) %>%
      mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
      select(year:day, dest, arr_delay, prop_delay)
```

```
# A tibble: 131,106 x 6
# Groups:   dest [77]
    year month   day dest  arr_delay prop_delay
   <int> <int> <int> <chr>     <dbl>      <dbl>
 1  2013     1     1 IAH          11  0.000111
 2  2013     1     1 IAH          20  0.000201
 3  2013     1     1 MIA          33  0.000235
 4  2013     1     1 ORD          12  0.0000424
 5  2013     1     1 FLL          19  0.0000938
 6  2013     1     1 ORD           8  0.0000283
 7  2013     1     1 LAX           7  0.0000344
 8  2013     1     1 DFW          31  0.000282
 9  2013     1     1 ATL          12  0.0000400
10  2013     1     1 DTW          16  0.000116
# ... with 131,096 more rows
```

A grouped filter is a grouped mutate followed by an ungrouped filter. I generally avoid them except for quick-and-dirty manipulations: otherwise, it's hard to check that you've done the manipulation correctly.

Functions that work most naturally in grouped mutates and filters are known as window functions (versus the summary functions used for summaries).