Data Preparation

Data preparation is the very first thing that you do and spend a lot of time on as a data analyst much before trying to build predictive models using that data.

In essence data preparation is all about processing data to get it ready for all kinds of analysis. All industry data collection is mostly driven by business process at front , not by the needs of predictive models. These various processes at some or the other point become reason for introduction of errors here and there in the data.

There can be many kind of reasons [not necessarily errors] for which we'd need to pre process our data and change it for better.

- Missing data
- Potentially incorrect data
- Need for changing form of the data

We'll discuss various reasons and methods to achieve our pre-processing goals going forward.

Handling Missing Values and Outliers

You'll figure out that treatment of both missing values and outliers can at times be very similar. Reason being , both kind of observations are basically not in a state to be used because of missing/ or miss information.

Treatment of missing values:

Removing observation with missing values

You can do that (with training data) if missing values are a very very small chunk of the data that you are dealing with. However you need to keep following things in mind while removing the observations because of missing data:

1. If observations with missing values are significant chunk of the data then you should not drop all observations with missing values
2. If the variable which had missing values has entered in your model, you need to plan what to do when you encounter missing values in the unseen data while model has been put in production.

Note: in general you wont be able to just ignore missing value when you make prediction in production (for test data). If your model makes use of a variable which has missing values , you'll have to figure out someway to impute the missing data

Imputing [filling up] missing values with mean/median of the respective variables.

You can simply fill missing values with central tendency measure of the data (mean/median)

Imputing with business logic

Many at times , we know what a missing value might mean in the context of business process. For example, If account balance is missing for the bank account , it might mean that the account balance is zero.

Note: Missing values for categorical data can be treated as just another category unless youy have sound business knowledge to replace them with some other pre-existing category

Treatment of Outliers:

Removing observations with outliers

There are two issues with including outliers in the predictive analysis

1. Because of otuliers , the predictor variables ranges get inflated artificially . The model that you get might not be applicable across that range
2. Some outliers have high leverage in context of the modelling process. In presence of such observations you'll get a model which is not a good fit for the general population [data].

If you are preparing data for predictive modelling , you need to remove outliers. However if the variable with outliers is present in the model, you need to figure out what to do when you encounter outlier values in the unseen data while model has been put in production.

Flooring/Capping

In some cases it might make sense to impute outlying values with upper and lower limits when they exceed either of these values. Imputing with lower limit is called flooring and imputing with upper limit is called capping.

Imputing with business logic

Many at times , we know what an outlier value might mean in the context of business process.

Note: ALgorithms like randomforest, boosting machines are pretty robust to noise in the data by design and you dont need to worry about doing this.

Need for changing form of the data

Transforming and extracting information from the existing data

Consider a simple transaction date and time column for an eCommerce website. A simple column containing dates will not be of much use but a lot of information can be extracted from this simple looking data. E.g. : Information regarding gaps between transactions, number of transactions happening every week or day or month etc.

Collapsing and Summarising Data:

Many at times we need to collapse data based on some grouping variables [This is more or less same as what we discussed in univariate statistics]. E.g. Finding out monthly summary of the data from a daily transaction data. In addition to tools which we learned in Univariate Statistics module we will learn few new things in the "Data Prep with R" section.

Reshaping Data

This is one of the very useful procedures we'll learn here. Below given is an example of long data

famid	year	faminc
1	96	40000

famid	year	faminc
1	97	40500
1	98	41000
2	96	45000
2	97	45400
2	98	45800
3	96	75000
3	98	77000

sometimes it'd make sense to this kind of the data into a wide format .Below given is an example of same data in a wide format.

famid	year_96	year_97	year_98
1	40000	40500	41000
2	45000	45400	45800
3	75000	.	77000

We'll learn how to achieve the same and more with `tidyr` package in R.

Data Preparation with R

Reading Data to R

Reading data in R is fairly simple. We'll be looking at function `read.csv` which helps you in reading data from the flat file formats. Flat files are files which you can open in a simple notepad and view the data. Excel files or any other proprietry data file format is NOT a flat file and can not be read using `read.csv`. Each properietry data file format has dedicated packages and associated functions for them . For example Excel files can be read using function `read.xlsx` found in the packaga `xlsx`

function `read.csv` comes with a lot options which you can see in the documentation. You dont need to pass values to all those options most of the time and set defaults work alright. However we are going to discuss few of them which you might use from time to time.

file : this is the name of the file to be read. In case file is in your working directory , only the file name is enough. If it is not in your working directory, you need to include entire path to the folder where file is along with the file name.

header : This is by default set to FALSE, if you set it to true, variable names are read and assigned from the first row of the file

sep : This is set to `,` by default. This tells R what symbol separates different columns in a row. For example a semi colon separated file has ";" as separator

row.names : You can pass a vector of row names if you want to set row names for your data. you generaly leave it as is

col.names : In case you want to force some other variable names you can pass those names as a vector to this option. Length of this vector or number of names that you are passing should match with number of columns. They are taken from first row of the file if header is not set to FALSE. You can set header to FALSE and R gives default col headers as `V1 V2 V3....`

stringsAsFactors : This is be default set to TRUE, you should always set this to FALSE while reading a flat file. What setting this to FALSE does that it imports character columns as character columns. You can later convert them to factors if you want; after pre processing the data.

na.strings : This is by default set to "NA". This means that any value which is written as "NA" will be assigned a missing/NA after reading. You can change this to other strings as well.

colClasses : By default this is set to NA or no forced classes. However you can pass a vector to force classes on the incoming columns. Without forcing , a column which contains only numbers or NA strings will be read as numeric. Column which contains even a single character value will be read as character [given that you have set stringsAsFactors to FALSE , otherwise it'll be stored as factors]

nrows : By default it is set to -1 which means all the rows from the file will be read. You can restrict that by passing a number smaller than the number of rows in file.

skip : By default this is set to 0, by assigning some number you can force R to skip first few rows of the file.

We will skip discussion on rest of the rarely used options. Lets look at one example. We'll be using function `read.csv`.

Remember if you are going to pass just file name you need to set your working directory to the folder which contains the file. You can do this by using function `setwd`. This is short for setting working directory.

```
setwd(" Here/Goes/Path/To/Your/Data/Folder/")
```

Also note that if you are working on a windows machine, you'd need to replace all "" in your path with"/" or "\".

you can check what is your current working directory by typing in `getwd()`

```
getwd()
```

We are going to import data file `bank-full.csv` here. Lets begin, we'll start with passing just the file name and let all other be option take their defaults. We'll change some as we come across issue with the imported data.

```
##   age.job.marital.education.default.balance.housing.loan.contact.day.month.duration.campaign.pdays.p
## 1                                    58;management;married;tertiary;no;2143;yes;no;unknown;5;may;261
## 2                                      44;technician;single;secondary;no;29;yes;no;unknown;5;may;151
```

```
bd=read.csv("bank-full.csv")
head(bd,2)
```

you can see that we have been fooled by the file extension and assumed that the separator for the data is comma where as in reality it is ";". Lets tell that to R by using option `sep`.

```
##   age        job marital education default balance housing loan contact
## 1  58 management married  tertiary      no    2143     yes   no unknown
## 2  44 technician  single secondary      no      29     yes   no unknown
##   day month duration campaign pdays previous poutcome  y
## 1   5   may      261        1    -1        0  unknown no
## 2   5   may      151        1    -1        0  unknown no
```

```
bd=read.csv("bank-full.csv",sep=";")
head(bd,2)
```

ok, this looks better. Now lets look at our data.

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.3.2
```

```r
glimpse(bd)
```

```
## Observations: 45,211
## Variables: 17
## $ age       <int> 58, 44, 33, 47, 33, 35, 28, 42, 58, 43, 41, 29, 53, ...
## $ job       <fct> management, technician, entrepreneur, blue-collar, u...
## $ marital   <fct> married, single, married, married, single, married, ...
## $ education <fct> tertiary, secondary, secondary, unknown, unknown, te...
## $ default   <fct> no, no, no, no, no, no, no, yes, no, no, no, no, no,...
## $ balance   <int> 2143, 29, 2, 1506, 1, 231, 447, 2, 121, 593, 270, 39...
## $ housing   <fct> yes, yes, yes, yes, no, yes, yes, yes, yes, yes, yes...
## $ loan      <fct> no, no, yes, no, no, no, yes, no, no, no, no, no, no...
## $ contact   <fct> unknown, unknown, unknown, unknown, unknown, unknown...
## $ day       <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5...
## $ month     <fct> may, may, may, may, may, may, may, may, may, may, ma...
## $ duration  <int> 261, 151, 76, 92, 198, 139, 217, 380, 50, 55, 222, 1...
## $ campaign  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ pdays     <int> -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, ...
## $ previous  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ poutcome  <fct> unknown, unknown, unknown, unknown, unknown, unknown...
## $ y         <fct> no, no, no, no, no, no, no, no, no, no, no, no, no, ...
```

you can see that all of our character columns have been stored as factors. This needs to be avoided. And we can do so by using option `stringsAsFactors`. Best of us make mistake by misspelling that option. Dont get frustrated. It'll become a parctice to make mistake, realise and correct.

```
## Observations: 45,211
## Variables: 17
## $ age       <int> 58, 44, 33, 47, 33, 35, 28, 42, 58, 43, 41, 29, 53, ...
## $ job       <chr> "management", "technician", "entrepreneur", "blue-co...
## $ marital   <chr> "married", "single", "married", "married", "single",...
## $ education <chr> "tertiary", "secondary", "secondary", "unknown", "un...
## $ default   <chr> "no", "no", "no", "no", "no", "no", "no", "yes", "no...
## $ balance   <int> 2143, 29, 2, 1506, 1, 231, 447, 2, 121, 593, 270, 39...
## $ housing   <chr> "yes", "yes", "yes", "yes", "no", "yes", "yes", "yes...
## $ loan      <chr> "no", "no", "yes", "no", "no", "no", "yes", "no", "n...
## $ contact   <chr> "unknown", "unknown", "unknown", "unknown", "unknown...
## $ day       <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5...
## $ month     <chr> "may", "may", "may", "may", "may", "may", "may", "ma...
## $ duration  <int> 261, 151, 76, 92, 198, 139, 217, 380, 50, 55, 222, 1...
## $ campaign  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ pdays     <int> -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, ...
## $ previous  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ poutcome  <chr> "unknown", "unknown", "unknown", "unknown", "unknown...
## $ y         <chr> "no", "no", "no", "no", "no", "no", "no", "no", "no"...
```

```r
bd=read.csv("bank-full.csv",sep=";",stringsAsFactors = FALSE)
glimpse(bd)
```

So thats taken care of , big relief. Next lets look at what values our variable job takes.

```r
table(bd$job)
```

```
##
##        admin.   blue-collar  entrepreneur     housemaid    management
##          5171          9732          1487          1240          9458
```

```
##       retired self-employed      services       student    technician 
##          2264          1579          4154           938          7597 
##    unemployed       unknown 
##          1303           288
```

you can see that there are 288 observations where the value is `unknown` , if you want you can set it to missing by using option `na.strings`. But remember this will set the value `unknown` as missing for all the columns. If you want to do it only for one of columns then do that **after** you have imported the data.

```
## [1] 288
```

```
bd=read.csv("bank-full.csv",sep=";",stringsAsFactors = FALSE,na.strings = "unknown")
sum(is.na(bd$job))
```

You can see that , now column `job` has 288 missing values. This was to show you how to use option `na.string`. In general it is not a good practice to set any random value as missing . So , for practice its alright, but dont set `unknown` to missing in general unless you have good reason to do so. In fact in many of the cases of categorical variables , `unknown` itself can be taken as a valid category as you'll realise later.

We dont need to change default values of other options for this importing. Same will be the case for you as well for most of the data. If it is not , feel free to use any of the option described above.

Apply Functions

We'll start with discussion on apply family of function which are very handy way to summarize as well as do other operations collectively on your data. Lets say we want to get means of all columns present in the data mtcars. We could achieve that writing a for loop across columns with function mean.

```
for(i in 1:ncol(mtcars)){
  print(mean(mtcars[,i]))
}
```

```
## [1] 20.09062
## [1] 6.1875
## [1] 230.7219
## [1] 146.6875
## [1] 3.596563
## [1] 3.21725
## [1] 17.84875
## [1] 0.4375
## [1] 0.40625
## [1] 3.6875
## [1] 2.8125
```

Fine you get the result, but its not in a very convenient format and code is not pretty. What if there exists a function which lets us do this without writing these loops and having to go through managing iterations for a function. This is such a common scenario in data processing, R has a family of dedicated funcitons for this. We'll first talk about `lapply`. This lets you apply a function repeatedly on a list/vector , the outcome is a list of results. Lets see an example.

```
x=round(rnorm(10),2)
x
```

```
##  [1]  0.15 -1.20  0.15 -0.12  0.32  0.70 -0.31  1.48 -1.47  0.54
```

```
lapply(x,log10)
```

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

```
## [1] -0.8239087
##
## [[2]]
## [1] NaN
##
## [[3]]
## [1] -0.8239087
##
## [[4]]
## [1] NaN
##
## [[5]]
## [1] -0.49485
##
## [[6]]
## [1] -0.154902
##
## [[7]]
## [1] NaN
##
## [[8]]
## [1] 0.1702617
##
## [[9]]
## [1] NaN
##
## [[10]]
## [1] -0.2676062
```

But above operation can be easily achieved using vector operations. Infact a simple `log(x)` will give you the same result and in a much usable format as well. But then what good is this `lapply` thing? Lets put it to a better use and with more options. How about , if you had a lot of text files in your folder and you wanted to import them all. One solution will be to write one line of `read.csv` for all these files or may be you can run a loop. Better you could pass all those names to function `read.csv` using `lapply`. Lets see

```
# Before running these codes , you'll have to set your
# working directory to the folder "namesbystate".
# You will find this folder inside "Data" folder
# which you downloaded from LMS
file_names=list.files(getwd(),pattern="*.TXT")
files=lapply(file_names,read.csv,header=F, stringsAsFactors = F)
```

See, we can pass other common options to function `read.csv` in `lapply`. File names in the object `file_names` are passed one by one to function `read.csv`. The output `files` is simply a list of data frames. If you want to combine them, you can do so by using function `do.call`.

```
file=do.call(rbind,files)
```

`do.call` here passes all the elements in the second argument to function mentioned in first argument.

In just three simple lines of code , we have read data from 50+ files and combined it into one. In just three lines! All thanks to `lapply`. At times the output in form of a list becomes difficult to handle. You can use `sapply` in these cases. `sapply` works exactly like `lapply`, only difference being, it tries to vectorise output of `lapply` [if it is possible].

```
sapply(x, log)
```

```
##  [1] -1.8971200        NaN -1.8971200        NaN -1.1394343 -0.3566749
##  [7]        NaN  0.3920421        NaN -0.6161861
```

Coming back to our first problem of getting mean for all columns . Yes you can use `lapply`, because data frames are nothing but list of vectors. There is another function in apply family named `apply` which provides better output and a little more funcitonality when you want to apply function iteratively on data frame elements.

```
apply(mtcars,2,mean)
```

```
##        mpg        cyl       disp         hp       drat         wt
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##       qsec         vs         am       gear       carb
##  17.848750   0.437500   0.406250   3.687500   2.812500
```

The first argument to `apply` is the name of the data frame. 3rd Argument is the function which is going to get applied. Second argument takes two values : [1 or 2]. 1 stands for `rows` and 2 for `columns`. Which means, if you put 2 in the second argument , function in the 3rd argument will be applied on the columns of the data frames. If the value is given as 1, function gets applied on rows. But it seems kind of odd functionality to provide, when will we really need to apply a function across rows? lets see an example. Here is a data which tells temperature recorded for a month , thrice a day.

```
##   days T1 T2 T3
## 1    1 20 30 23
## 2    2 28 26 21
## 3    3 27 21 27
## 4    4 24 22 24
## 5    5 30 24 27
## 6    6 26 30 24
```

How to add a column to data with max temperature for the day?

```
temps$max_temp=apply(temps,1,max)
head(temps)
```

```
##   days T1 T2 T3 max_temp
## 1    1 20 30 23       30
## 2    2 28 26 21       28
## 3    3 27 21 27       27
## 4    4 24 22 24       24
## 5    5 30 24 27       30
## 6    6 26 30 24       30
```

Functions whic you pass to apply are not limited to pre existing function in R. You can write your own and pass it to apply family functions. Lets write a function which returns uuper limit of outliers given a variable column [vector]. $\mu + 3 * \sigma$

```
outlier_upper=function(x){
  m=mean(x);
  s=sd(x);
  return(m+3*s);
}
apply(mtcars,2,outlier_upper)
```

```
##        mpg        cyl       disp         hp       drat         wt
##  38.171469  11.545265 602.537956 352.376105   5.200599   6.152622
##       qsec         vs         am       gear       carb
##  23.209580   1.949548   1.903223   5.900912   7.658100
```

you can even write on the fly functions. What you need to remember here is what goes as input to the function. In case of `apply` , input is the entire row or column. Lets use `apply` to find out how many outliers each column has according to function `oulier_upper`.

```
apply(mtcars,2,function(x) sum(x>outlier_upper(x)))
```

```
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##    0    0    0    0    0    0    0    0    0    0    1
```

This is all good, but what if i want to get a group wise summary of any variable. `tapply` comes to your rescue. First arguement to `tapply` is the column for which we are looking for summary, second argument is the grouping variable, 3rd argument is the function which will be applied on the groups.

```
tapply(mtcars$mpg,mtcars$am,mean)
```

```
##        0        1
## 17.14737 24.39231
```

For getting group wise summary of all the variable in the dataset mtcars you can use a combination of `apply` and `tapply`.

```
apply(mtcars,2,function(x) tapply(x,mtcars$am,mean))
```

```
##        mpg      cyl     disp       hp     drat       wt     qsec        vs
## 0 17.14737 6.947368 290.3789 160.2632 3.286316 3.768895 18.18316 0.3684211
## 1 24.39231 5.076923 143.5308 126.8462 4.050000 2.411000 17.36000 0.5384615
##   am     gear     carb
## 0  0 3.210526 2.736842
## 1  1 4.384615 2.923077
```

I am leaving function `mapply` for you to explore on your own. ### Useful function for Data Prep: ifelse

Creating variables/vectors with simple algebraic operations is straight forward. Just as a recap lets add one variable to data frame Arthritis.

```
library(vcd)
Arthritis$new=log(Arthritis$Age)
head(Arthritis)
```

```
##   ID Treatment  Sex Age Improved      new
## 1 57   Treated Male  27     Some 3.295837
## 2 46   Treated Male  29     None 3.367296
## 3 77   Treated Male  30     None 3.401197
## 4 17   Treated Male  32   Marked 3.465736
## 5 36   Treated Male  46   Marked 3.828641
## 6 23   Treated Male  58   Marked 4.060443
```

What if we wanted to create an indicator variable which takes value 0 or 1 according to Age being less than or greater than 40? These simple algebraic operations will not work. We'll have to use conditional operators.

```
Arthritis$new=as.numeric(Arthritis$Age<40)
head(Arthritis)
```

```
##   ID Treatment  Sex Age Improved new
## 1 57   Treated Male  27     Some   1
## 2 46   Treated Male  29     None   1
## 3 77   Treated Male  30     None   1
## 4 17   Treated Male  32   Marked   1
## 5 36   Treated Male  46   Marked   0
## 6 23   Treated Male  58   Marked   0
```

This seems trivial too, now what if we want Age to be floored to 40 whenever it is less than 40 and other wise kept as it is. We wont be able to achieve this with simple conditional statement either. We will be using function `ifelse`to achieve the same.

```
x=sample(40,10)
x
```

```
##  [1] 32 10 16 26 23  3 24 36 27 19
```

```
y=ifelse(x>20,20,x)
y
```

```
##  [1] 20 10 16 20 20  3 20 20 20 19
```

Now lets use this to add a variable in the data frame

```
Arthritis$new=ifelse(Arthritis$Age<40,40,Arthritis$Age)
head(Arthritis)
```

```
##   ID Treatment  Sex Age Improved new
## 1 57   Treated Male  27     Some  40
## 2 46   Treated Male  29     None  40
## 3 77   Treated Male  30     None  40
## 4 17   Treated Male  32   Marked  40
## 5 36   Treated Male  46   Marked  46
## 6 23   Treated Male  58   Marked  58
```

Another function that i wanted to discuss here is lag, this can be used to create lag counter parts to your data. Look at this monthly sales data.

```
##    months sales
## 1       1  1879
## 2       2  1556
## 3       3  1082
## 4       4  1660
## 5       5  1315
## 6       6  1537
## 7       7  1798
## 8       8  1261
## 9       9  1021
## 10     10  1979
## 11     11  1388
## 12     12  1592
```

If you were asked to get month on month growth you can achieve it by differencing with lag counter part of sales. Lets see how to do it.

Grammar of Data Wrangling : dplyr

We have seen ways to modify and summarise data in base R. Again those functionalities are kind of scattered and not streamlined. If you think about it you can achieve almost all kind of modifications to data using these verbs:

- filter : conditional filtering of data
- select : selecting columns
- mutate : adding/modifying columns
- arrange : sorting columns
- summarise (with adverb group_by) : Collapsing Data to its summaries

Package `dplyr` comes with these verbs for data wrangling. Next we'll see how to achieve different data wrangling task in base R and the same in `dplyr`. Of course `dplyr` comes with some addtional fuctionlaities too and we'll be looking at those as well. Before you start , install packages `dplyr` and `hflights`. We'll be using data set `hflights`. You can get details of the data `hflights` after you have loaded library `hflights` by typing `?hflights`.

```r
library(dplyr)
library(hflights)

data(hflights)
head(hflights)
```

```
##      Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
## 5424 2011     1          1         6    1400    1500            AA
## 5425 2011     1          2         7    1401    1501            AA
## 5426 2011     1          3         1    1352    1502            AA
## 5427 2011     1          4         2    1403    1513            AA
## 5428 2011     1          5         3    1405    1507            AA
## 5429 2011     1          6         4    1359    1503            AA
##      FlightNum TailNum ActualElapsedTime AirTime ArrDelay DepDelay Origin
## 5424       428  N576AA                60      40      -10        0    IAH
## 5425       428  N557AA                60      45       -9        1    IAH
## 5426       428  N541AA                70      48       -8       -8    IAH
## 5427       428  N403AA                70      39        3        3    IAH
## 5428       428  N492AA                62      44       -3        5    IAH
## 5429       428  N262AA                64      45       -7       -1    IAH
##      Dest Distance TaxiIn TaxiOut Cancelled CancellationCode Diverted
## 5424  DFW      224      7      13         0                         0
## 5425  DFW      224      6       9         0                         0
## 5426  DFW      224      5      17         0                         0
## 5427  DFW      224      9      22         0                         0
## 5428  DFW      224      9       9         0                         0
## 5429  DFW      224      6      13         0                         0
```

We'll start with our first function `tbl_df` which converts a data.frame to a tabular format for which display on console is better. It changes nothing else about the data frame

```r
flights=tbl_df(hflights)
flights
```

```
## # A tibble: 227,496 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##  * <int> <int>      <int>     <int>   <int>   <int> <chr>
##  1  2011     1          1         6    1400    1500 AA
##  2  2011     1          2         7    1401    1501 AA
##  3  2011     1          3         1    1352    1502 AA
##  4  2011     1          4         2    1403    1513 AA
##  5  2011     1          5         3    1405    1507 AA
##  6  2011     1          6         4    1359    1503 AA
##  7  2011     1          7         5    1359    1509 AA
##  8  2011     1          8         6    1355    1454 AA
##  9  2011     1          9         7    1443    1554 AA
## 10  2011     1         10         1    1443    1553 AA
## # ... with 227,486 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
```

```
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

Lets look at condition filtering of the data. We'll start with base R approach to view all flights on January 1

```r
flights[flights$Month==1 & flights$DayofMonth==1, ]
```

```
## # A tibble: 552 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##    <int> <int>      <int>     <int>   <int>   <int> <chr>
##  1  2011     1          1         6    1400    1500 AA
##  2  2011     1          1         6     728     840 AA
##  3  2011     1          1         6    1631    1736 AA
##  4  2011     1          1         6    1756    2112 AA
##  5  2011     1          1         6    1012    1347 AA
##  6  2011     1          1         6    1211    1325 AA
##  7  2011     1          1         6     557     906 AA
##  8  2011     1          1         6    1824    2106 AS
##  9  2011     1          1         6     654    1124 B6
## 10  2011     1          1         6    1639    2110 B6
## # ... with 542 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

dplyr approach:

```r
#note: you can use comma or ampersand to represent AND condition
filter(flights, Month==1, DayofMonth==1)
```

```
## Warning: package 'bindrcpp' was built under R version 3.3.2

## # A tibble: 552 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##    <int> <int>      <int>     <int>   <int>   <int> <chr>
##  1  2011     1          1         6    1400    1500 AA
##  2  2011     1          1         6     728     840 AA
##  3  2011     1          1         6    1631    1736 AA
##  4  2011     1          1         6    1756    2112 AA
##  5  2011     1          1         6    1012    1347 AA
##  6  2011     1          1         6    1211    1325 AA
##  7  2011     1          1         6     557     906 AA
##  8  2011     1          1         6    1824    2106 AS
##  9  2011     1          1         6     654    1124 B6
## 10  2011     1          1         6    1639    2110 B6
## # ... with 542 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

```r
# use pipe for OR condition
filter(flights, UniqueCarrier=="AA" | UniqueCarrier=="UA")
```

```
## # A tibble: 5,316 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##    <int> <int>      <int>     <int>   <int>   <int> <chr>
```

```
## 1  2011     1          1         6    1400    1500 AA
## 2  2011     1          2         7    1401    1501 AA
## 3  2011     1          3         1    1352    1502 AA
## 4  2011     1          4         2    1403    1513 AA
## 5  2011     1          5         3    1405    1507 AA
## 6  2011     1          6         4    1359    1503 AA
## 7  2011     1          7         5    1359    1509 AA
## 8  2011     1          8         6    1355    1454 AA
## 9  2011     1          9         7    1443    1554 AA
## 10 2011     1         10         1    1443    1553 AA
## # ... with 5,306 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

```r
# you can also use %in% operator
filter(flights, UniqueCarrier %in% c("AA", "UA"))
```

```
## # A tibble: 5,316 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##    <int> <int>      <int>     <int>   <int>   <int> <chr>
## 1  2011     1          1         6    1400    1500 AA
## 2  2011     1          2         7    1401    1501 AA
## 3  2011     1          3         1    1352    1502 AA
## 4  2011     1          4         2    1403    1513 AA
## 5  2011     1          5         3    1405    1507 AA
## 6  2011     1          6         4    1359    1503 AA
## 7  2011     1          7         5    1359    1509 AA
## 8  2011     1          8         6    1355    1454 AA
## 9  2011     1          9         7    1443    1554 AA
## 10 2011     1         10         1    1443    1553 AA
## # ... with 5,306 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

See, you don't need to bother with that $ reference to data frame all the time. Code is much neater and readable. Lets look at column selection dropping by name. You'll be definietly pleseantly surprised by the additional functionalities by `dplyr`.

```r
# base R approach to select DepTime, ArrTime, and FlightNum columns
flights[, c("DepTime", "ArrTime", "FlightNum")]
```

```
## # A tibble: 227,496 x 3
##    DepTime ArrTime FlightNum
##      <int>   <int>     <int>
## 1     1400    1500       428
## 2     1401    1501       428
## 3     1352    1502       428
## 4     1403    1513       428
## 5     1405    1507       428
## 6     1359    1503       428
## 7     1359    1509       428
## 8     1355    1454       428
```

```
##  9    1443    1554       428
## 10    1443    1553       428
## # ... with 227,486 more rows
```

```r
# dplyr approach
select(flights, DepTime, ArrTime, FlightNum)
```

```
## # A tibble: 227,496 x 3
##    DepTime ArrTime FlightNum
##  *   <int>   <int>     <int>
##  1    1400    1500       428
##  2    1401    1501       428
##  3    1352    1502       428
##  4    1403    1513       428
##  5    1405    1507       428
##  6    1359    1503       428
##  7    1359    1509       428
##  8    1355    1454       428
##  9    1443    1554       428
## 10    1443    1553       428
## # ... with 227,486 more rows
```

Use colon to select multiple contiguous columns, and use `contains` to match columns by name note: "starts_with", "ends_with" can also be used to match columns by name

```r
select(flights, Year:DayofMonth, contains("Taxi"), contains("Delay"))
```

```
## # A tibble: 227,496 x 7
##     Year Month DayofMonth TaxiIn TaxiOut ArrDelay DepDelay
##  * <int> <int>      <int>  <int>   <int>    <int>    <int>
##  1  2011     1          1      7      13      -10        0
##  2  2011     1          2      6       9      - 9        1
##  3  2011     1          3      5      17      - 8      - 8
##  4  2011     1          4      9      22        3        3
##  5  2011     1          5      9       9      - 3        5
##  6  2011     1          6      6      13      - 7      - 1
##  7  2011     1          7     12      15      - 1      - 1
##  8  2011     1          8      7      12      -16      - 5
##  9  2011     1          9      8      22       44       43
## 10  2011     1         10      6      19       43       43
## # ... with 227,486 more rows
```

you can drop variable by simply putting a `-` sign in front of the variable name.

Now what if we wanted to do many operations at once; for example, selction and conditional filtering. We can do so by nesting our functions.

```r
# nesting method to select UniqueCarrier and
# DepDelay columns and filter for delays over 60 minutes
filter(select(flights, UniqueCarrier, DepDelay), DepDelay > 60)
```

```
## # A tibble: 10,242 x 2
##    UniqueCarrier DepDelay
##    <chr>            <int>
##  1 AA                  90
##  2 AA                  67
##  3 AA                  74
##  4 AA                 125
```

```
##   5 AA                    82
##   6 AA                    99
##   7 AA                    70
##   8 AA                    61
##   9 AA                    74
##  10 AS                    73
## # ... with 10,232 more rows
```

This nesting methodology becomes very cumbersome very fast . This defies the purpose with which we started , making our code more readable. Comes to your rescue `%>%` operator , also called chaining operator. You can read it as `then`. Basically when you use this operator , every subsequent line of code inherits inputs from the previous line. You'll be able to better understand this with the following example. Later on we'll rewrite the above nested code with the chaining operator.

```
x=sample(10,6)
x %>%
  log() %>%
  sum()
```

```
## [1] 7.783224
```

See, you don't have to pass any input to those functions , x goes as input to `log` and then modified x as `log(x)` goes as input to `sum`. Lets see how we can use this to rewrite the nested function that we saw above.

```
# chaining method
flights %>%
  select(UniqueCarrier, DepDelay) %>%
  filter(DepDelay > 60)
```

```
## # A tibble: 10,242 x 2
##    UniqueCarrier DepDelay
##    <chr>            <int>
##   1 AA                    90
##   2 AA                    67
##   3 AA                    74
##   4 AA                   125
##   5 AA                    82
##   6 AA                    99
##   7 AA                    70
##   8 AA                    61
##   9 AA                    74
##  10 AS                    73
## # ... with 10,232 more rows
```

See , no need to nest or keep on giving data reference for every operation. Isn't that neat!!

Next we move to ordering/sorting our data by using verb `arrange`.

```
# base R approach to select UniqueCarrier
# and DepDelay columns and sort by DepDelay
flights[order(flights$DepDelay), c("UniqueCarrier", "DepDelay")]
```

```
## # A tibble: 227,496 x 2
##    UniqueCarrier DepDelay
##    <chr>            <int>
##   1 OO                   -33
##   2 MQ                   -23
##   3 XE                   -19
```

```
##   4 XE                    -19
##   5 CO                    -18
##   6 EV                    -18
##   7 XE                    -17
##   8 CO                    -17
##   9 XE                    -17
##  10 MQ                    -17
## # ... with 227,486 more rows
```

```
# dplyr approach
flights %>%
  select(UniqueCarrier, DepDelay) %>%
  arrange(DepDelay)
```

```
## # A tibble: 227,496 x 2
##    UniqueCarrier DepDelay
##    <chr>            <int>
##   1 OO                    -33
##   2 MQ                    -23
##   3 XE                    -19
##   4 XE                    -19
##   5 CO                    -18
##   6 EV                    -18
##   7 XE                    -17
##   8 CO                    -17
##   9 XE                    -17
##  10 MQ                    -17
## # ... with 227,486 more rows
```

Next step is , your introduction to mutate or modifying/adding data to existing data.

```
# base R approach to create a new variable Speed (in mph)
flights$Speed <- flights$Distance / flights$AirTime*60
flights[, c("Distance", "AirTime", "Speed")]
```

```
## # A tibble: 227,496 x 3
##    Distance AirTime Speed
##       <int>   <int> <dbl>
##   1       224      40   336
##   2       224      45   299
##   3       224      48   280
##   4       224      39   345
##   5       224      44   305
##   6       224      45   299
##   7       224      43   313
##   8       224      40   336
##   9       224      41   328
##  10       224      45   299
## # ... with 227,486 more rows
```

```
# dplyr approach
flights %>%
  select(Distance, AirTime) %>%
  mutate(Speed = Distance/AirTime*60)
```

```
## # A tibble: 227,496 x 3
##    Distance AirTime Speed
```

```
##       <int>   <int> <dbl>
##  1      224      40   336
##  2      224      45   299
##  3      224      48   280
##  4      224      39   345
##  5      224      44   305
##  6      224      45   299
##  7      224      43   313
##  8      224      40   336
##  9      224      41   328
## 10      224      45   299
## # ... with 227,486 more rows
```

What we have been doing is getting the output to display, if you wanted to save it could do as we usually do in R. Say, we wanted to save above output to some data frame.

```
flight_sub=flights %>%
  select(Distance, AirTime) %>%
  mutate(Speed = Distance/AirTime*60)
```

We are done with data wrangling without collapsing it. Next we look at exactly that, summarising data by groups or collapsing data to its group wise summaries using dplyr.

```
# dplyr approach: create a table grouped by Dest,
# and then summarise each group by taking the mean of ArrDelay
flights %>%
  group_by(Dest) %>%
  summarise(avg_delay = mean(ArrDelay, na.rm=TRUE))
```

```
## # A tibble: 116 x 2
##    Dest  avg_delay
##    <chr>     <dbl>
##  1 ABQ        7.23
##  2 AEX        5.84
##  3 AGS        4.00
##  4 AMA        6.84
##  5 ANC       26.1
##  6 ASE        6.79
##  7 ATL        8.23
##  8 AUS        7.45
##  9 AVL        9.97
## 10 BFL      -13.2
## # ... with 106 more rows
```

This pretty much finishes our discussion on dplyr verbs and adverbs. I have given few more examples to learn new useful functionalities which we havent introduced yet.

```
# for each day of the year, count the total number of flights
# and sort in descending order
z=flights %>%
  group_by(Month, DayofMonth) %>%
  summarise(flight_count = n()) %>%
  arrange(desc(flight_count))


# rewrite more simply with the 'tally' function
flights %>%
```

```r
  group_by(Month, DayofMonth) %>%
  tally(sort = TRUE)
```

```
## # A tibble: 365 x 3
## # Groups:   Month [12]
##    Month DayofMonth     n
##    <int>      <int> <int>
##  1     8          4   706
##  2     8         11   706
##  3     8         12   706
##  4     8          5   705
##  5     8          3   704
##  6     8         10   704
##  7     1          3   702
##  8     7          7   702
##  9     7         14   702
## 10     7         28   701
## # ... with 355 more rows
```

```r
# for each destination, count the total number of flights
# and the number of distinct planes that flew there
flights %>%
  group_by(Dest) %>%
  summarise(flight_count = n(), plane_count = n_distinct(TailNum))
```

```
## # A tibble: 116 x 3
##    Dest  flight_count plane_count
##    <chr>        <int>       <int>
##  1 ABQ           2812         716
##  2 AEX            724         215
##  3 AGS              1           1
##  4 AMA           1297         158
##  5 ANC            125          38
##  6 ASE            125          60
##  7 ATL           7886         983
##  8 AUS           5022        1015
##  9 AVL            350         142
## 10 BFL            504          70
## # ... with 106 more rows
```

```r
# for each destination, show the number of cancelled
# and not cancelled flights
flights %>%
  group_by(Dest) %>%
  select(Cancelled) %>%
  table() %>%
  head()
```

```
## Adding missing grouping variables: `Dest`
```

```
##      Cancelled
## Dest     0    1
##   ABQ 2787   25
##   AEX  712   12
##   AGS    1    0
##   AMA 1265   32
```

```
##   ANC  125    0
##   ASE  120    5
```

```r
# for each month, calculate the number of flights
# and the change from the previous month
flights %>%
  group_by(Month) %>%
  summarise(flight_count = n()) %>%
  mutate(change = flight_count - lag(flight_count))
```

```
## # A tibble: 12 x 3
##    Month flight_count change
##    <int>        <int>  <int>
##  1     1        18910     NA
##  2     2        17128  -1782
##  3     3        19470   2342
##  4     4        18593  - 877
##  5     5        19172    579
##  6     6        19600    428
##  7     7        20548    948
##  8     8        20176  - 372
##  9     9        18065  -2111
## 10    10        18696    631
## 11    11        18021  - 675
## 12    12        19117   1096
```

```r
# rewrite more simply with the `tally` function
flights %>%
  group_by(Month) %>%
  tally() %>%
  mutate(change = n - lag(n))
```

```
## # A tibble: 12 x 3
##    Month     n change
##    <int> <int>  <int>
##  1     1 18910     NA
##  2     2 17128  -1782
##  3     3 19470   2342
##  4     4 18593  - 877
##  5     5 19172    579
##  6     6 19600    428
##  7     7 20548    948
##  8     8 20176  - 372
##  9     9 18065  -2111
## 10    10 18696    631
## 11    11 18021  - 675
## 12    12 19117   1096
```

```r
# base R approach to view the structure of an object
str(flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    227496 obs. of  22 variables:
##  $ Year             : int  2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 ...
##  $ Month            : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ DayofMonth       : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ DayOfWeek        : int  6 7 1 2 3 4 5 6 7 1 ...
```

```
##  $ DepTime          : int  1400 1401 1352 1403 1405 1359 1359 1355 1443 1443 ...
##  $ ArrTime          : int  1500 1501 1502 1513 1507 1503 1509 1454 1554 1553 ...
##  $ UniqueCarrier    : chr  "AA" "AA" "AA" "AA" ...
##  $ FlightNum        : int  428 428 428 428 428 428 428 428 428 428 ...
##  $ TailNum          : chr  "N576AA" "N557AA" "N541AA" "N403AA" ...
##  $ ActualElapsedTime: int  60 60 70 70 62 64 70 59 71 70 ...
##  $ AirTime          : int  40 45 48 39 44 45 43 40 41 45 ...
##  $ ArrDelay         : int  -10 -9 -8 3 -3 -7 -1 -16 44 43 ...
##  $ DepDelay         : int  0 1 -8 3 5 -1 -1 -5 43 43 ...
##  $ Origin           : chr  "IAH" "IAH" "IAH" "IAH" ...
##  $ Dest             : chr  "DFW" "DFW" "DFW" "DFW" ...
##  $ Distance         : int  224 224 224 224 224 224 224 224 224 224 ...
##  $ TaxiIn           : int  7 6 5 9 9 6 12 7 8 6 ...
##  $ TaxiOut          : int  13 9 17 22 9 13 15 12 22 19 ...
##  $ Cancelled        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ CancellationCode : chr  "" "" "" "" ...
##  $ Diverted         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Speed            : num  336 299 280 345 305 ...
```

```r
# dplyr approach: better formatting, and adapts to your screen width
glimpse(flights)
```

```
## Observations: 227,496
## Variables: 22
## $ Year              <int> 2011, 2011, 2011, 2011, 2011, 2011, 2011, 20...
## $ Month             <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ DayofMonth        <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1...
## $ DayOfWeek         <int> 6, 7, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6,...
## $ DepTime           <int> 1400, 1401, 1352, 1403, 1405, 1359, 1359, 13...
## $ ArrTime           <int> 1500, 1501, 1502, 1513, 1507, 1503, 1509, 14...
## $ UniqueCarrier     <chr> "AA", "AA", "AA", "AA", "AA", "AA", "AA", "A...
## $ FlightNum         <int> 428, 428, 428, 428, 428, 428, 428, 428, 428,...
## $ TailNum           <chr> "N576AA", "N557AA", "N541AA", "N403AA", "N49...
## $ ActualElapsedTime <int> 60, 60, 70, 70, 62, 64, 70, 59, 71, 70, 70, ...
## $ AirTime           <int> 40, 45, 48, 39, 44, 45, 43, 40, 41, 45, 42, ...
## $ ArrDelay          <int> -10, -9, -8, 3, -3, -7, -1, -16, 44, 43, 29,...
## $ DepDelay          <int> 0, 1, -8, 3, 5, -1, -1, -5, 43, 43, 29, 19, ...
## $ Origin            <chr> "IAH", "IAH", "IAH", "IAH", "IAH", "IAH", "I...
## $ Dest              <chr> "DFW", "DFW", "DFW", "DFW", "DFW", "DFW", "D...
## $ Distance          <int> 224, 224, 224, 224, 224, 224, 224, 224, 224,...
## $ TaxiIn            <int> 7, 6, 5, 9, 9, 6, 12, 7, 8, 6, 8, 4, 6, 5, 6...
## $ TaxiOut           <int> 13, 9, 17, 22, 9, 13, 15, 12, 22, 19, 20, 11...
## $ Cancelled         <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ CancellationCode  <chr> "", "", "", "", "", "", "", "", "", "", "", ...
## $ Diverted          <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ Speed             <dbl> 336.0000, 298.6667, 280.0000, 344.6154, 305....
```

Sampling Your Data

As moving slowly towards predictive modelling , you'd need to take randome sample from your data for different purposes. You can achieve that in a rather simple manner by using the function `sample` which we have used a lot so far. Here goes an example for taking 70% random data from data frame mtcars.

```r
set.seed(1)
s=sample(1:nrow(mtcars),0.7*nrow(mtcars))
# we are using set.seed for our random sample to be reproducible
```

You can now use this vector `s` as row index vector to take sample data from the data frame.

```r
mtcars_sample=mtcars[s,]
```

How do I get the rest of the observations which are not in sample taken above? simple:

```r
mtcars_remaining=mtcars[-s,]
```

How to randomly bootstrap your data? Again, you can achieve that by using sampling with replacement with function `sample`

```r
set.seed(1)
s=sample(1:nrow(mtcars),100,replace = TRUE)
mtcars_bootstrapped=mtcars[s,]
```

Next we'll look into reshaping your data.

Reshaping Your Data : tidyr

Reshaping your data with tidyr requires two simple tasks `gather` and `seperate`. It also has counterparts to these which are `spread` and `unite`. Lets look at few examples to understand what we can achieve with this.

Look at this data. This data correspond to three peopel being given two different drugs and then their heart rate being recorded. heart rate corresponding to each drug is put in their respective columns named a and b:

```
##      name  a  b
## 1  Wilbur 67 56
## 2 Petunia 80 90
## 3 Gregory 64 50
```

Now what if you want all the heart rates , instead of being spread across multiple columns, gathered into a single column. We can do that using the `gather` function.

```r
library(tidyr)

messy %>%
  gather(drug,heartrate,a:b)
```

```
##      name drug heartrate
## 1  Wilbur    a        67
## 2 Petunia    a        80
## 3 Gregory    a        64
## 4  Wilbur    b        56
## 5 Petunia    b        90
## 6 Gregory    b        50
```

What this does is , it gathers column names from a to b in the new variable `drug` and the column `heartrate` contains values which were in wide format previously in the data in columns a to b.

Lets look at another example which will let us look more into function `gather` and we'll also see where we can use function `separate`. The data set that we start with is this

```
##   id       trt    work.T1   home.T1   work.T2    home.T2
## 1  1 treatment 0.08513597 0.6158293 0.1135090 0.05190332
## 2  2   control 0.22543662 0.4296715 0.5959253 0.26417767
```

```
## 3  3 treatment 0.27453052 0.6516557 0.3580500 0.39879073
## 4  4   control 0.27230507 0.5677378 0.4288094 0.83613414
```

```
tidier = messy %>%
  gather(key,time,-id,-trt)
tidier
```

```
##    id       trt     key       time
## 1   1 treatment work.T1 0.08513597
## 2   2   control work.T1 0.22543662
## 3   3 treatment work.T1 0.27453052
## 4   4   control work.T1 0.27230507
## 5   1 treatment home.T1 0.61582931
## 6   2   control home.T1 0.42967153
## 7   3 treatment home.T1 0.65165567
## 8   4   control home.T1 0.56773775
## 9   1 treatment work.T2 0.11350898
## 10  2   control work.T2 0.59592531
## 11  3 treatment work.T2 0.35804998
## 12  4   control work.T2 0.42880942
## 13  1 treatment home.T2 0.05190332
## 14  2   control home.T2 0.26417767
## 15  3 treatment home.T2 0.39879073
## 16  4   control home.T2 0.83613414
```

Here after second argument `-id,-trt` tells gather to use rest of the variable for gathering. which are work.T1, home.T1, work.T2,home.T2.

Now if we want to convert this variable `key` in to two separate variables because of the information contained we can do that by using function `separate`.

```
tidier=tidier %>% separate(key,into=c("location","shift"),sep="\\.")
tidier
```

```
##    id       trt location shift       time
## 1   1 treatment     work    T1 0.08513597
## 2   2   control     work    T1 0.22543662
## 3   3 treatment     work    T1 0.27453052
## 4   4   control     work    T1 0.27230507
## 5   1 treatment     home    T1 0.61582931
## 6   2   control     home    T1 0.42967153
## 7   3 treatment     home    T1 0.65165567
## 8   4   control     home    T1 0.56773775
## 9   1 treatment     work    T2 0.11350898
## 10  2   control     work    T2 0.59592531
## 11  3 treatment     work    T2 0.35804998
## 12  4   control     work    T2 0.42880942
## 13  1 treatment     home    T2 0.05190332
## 14  2   control     home    T2 0.26417767
## 15  3 treatment     home    T2 0.39879073
## 16  4   control     home    T2 0.83613414
```

Now lets use function `spread` and `unite` to roll this back to original format.

```
step1= tidier %>%
  unite(key,location,shift,sep=".")
step1
```

```
##    id       trt     key       time
## 1   1 treatment work.T1 0.08513597
## 2   2   control work.T1 0.22543662
## 3   3 treatment work.T1 0.27453052
## 4   4   control work.T1 0.27230507
## 5   1 treatment home.T1 0.61582931
## 6   2   control home.T1 0.42967153
## 7   3 treatment home.T1 0.65165567
## 8   4   control home.T1 0.56773775
## 9   1 treatment work.T2 0.11350898
## 10  2   control work.T2 0.59592531
## 11  3 treatment work.T2 0.35804998
## 12  4   control work.T2 0.42880942
## 13  1 treatment home.T2 0.05190332
## 14  2   control home.T2 0.26417767
## 15  3 treatment home.T2 0.39879073
## 16  4   control home.T2 0.83613414
```

```
step2=step1 %>%
  spread(key,time)
step2
```

```
##   id       trt   home.T1    home.T2    work.T1   work.T2
## 1  1 treatment 0.6158293 0.05190332 0.08513597 0.1135090
## 2  2   control 0.4296715 0.26417767 0.22543662 0.5959253
## 3  3 treatment 0.6516557 0.39879073 0.27453052 0.3580500
## 4  4   control 0.5677378 0.83613414 0.27230507 0.4288094
```

Usage of `unite` is straight forward, i will not discuss that in detail. What spread is doing that it makes column names by different values of variable `key` and then fills in those columns values from variable `time`.

Ofcourse these observations need to be uniquely identifiable with the help of other variables in the data.

Working with Date & Time Data in R

Lastly I'd like to cover a very imprtant topic in data prep that is to handle date time data . Why the special attention? Because the way date time data is recorded is essentially as a character and when it is dealt with, its like numbers . Now parsing these seemingly character strings to numbers in a way that they can represent data time and then throw those time zones in the mix and you have pretty difficult situation to handle. Fret not , we have package `lubridate` to make our life pretty easy.

Please install package `lubridate` before you begin running these codes.

Note: All our examples are based on converting character strings to dates using functions. When you are reading data from a file, read a date time column as character and convert it to a date time object type later. Its always easier to do.

We'll start with converting various dates in different formats stored as characters. These formats can be different in terms of with in that data in what order day, month and year appear.

Identify the order in which the year, month, and day appears in your dates. Now arrange "y", "m", and "d" in the same order. This is the name of the function in lubridate that will parse your dates. For example,

```
library(lubridate)
ymd("20110604")
```

```
## [1] "2011-06-04"
```

```r
mdy("06-04-2011")
```

```
## [1] "2011-06-04"
```

```r
dmy("04/06/2011")
```

```
## [1] "2011-06-04"
```

you include time components and timezones as well by simply adding the order of time components hours ("h"), minutes ("m") and seconds ("s"). Appropriate function name exists.

```r
arrive = ymd_hms("2011-06-04 12:00:00", tz = "Pacific/Auckland")

leave = ymd_hms("2011-08-10 14:00:00", tz = "Pacific/Auckland")
```

you can extract and set individual elements of the date as well.

```r
second(arrive)
```

```
## Warning in as.POSIXlt.POSIXct(x, tz = tz(x)): unknown timezone 'zone/tz/
## 2018c.1.0/zoneinfo/Asia/Kolkata'

## [1] 0
```

```r
second(arrive) = 25
```

```
## Warning in as.POSIXlt.POSIXct(x, tz = tz(x)): unknown timezone 'zone/tz/
## 2018c.1.0/zoneinfo/Asia/Kolkata'

## Warning in as.POSIXlt.POSIXct(date): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## Warning in as.POSIXct.POSIXlt(object): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## Warning in as.POSIXlt.POSIXct(new): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## Warning in as.POSIXct.POSIXlt(new): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'
```

```r
arrive
```

```
## Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## [1] "2011-06-04 12:00:25 NZST"
```

```r
second(arrive) = 0
```

```
## Warning in as.POSIXlt.POSIXct(x, tz = tz(x)): unknown timezone 'zone/tz/
## 2018c.1.0/zoneinfo/Asia/Kolkata'

## Warning in as.POSIXlt.POSIXct(date): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## Warning in as.POSIXct.POSIXlt(object): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## Warning in as.POSIXlt.POSIXct(new): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'

## Warning in as.POSIXct.POSIXlt(new): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'
```

```r
wday(arrive)
```

```
## Warning in as.POSIXlt.POSIXct(x, tz = tz(x)): unknown timezone 'zone/tz/
## 2018c.1.0/zoneinfo/Asia/Kolkata'
```

```
## [1] 7
```

```r
wday(arrive, label = TRUE)
```

```
## Warning in as.POSIXlt.POSIXct(x, tz = tz(x)): unknown timezone 'zone/tz/
## 2018c.1.0/zoneinfo/Asia/Kolkata'
```

```
## [1] Sat
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

I'm leaving for you to find functions for rest of the elements. They are there and names aren't difficult to figure out.

Time zones is something which can cause significant amount of frustration. This has been made very simple in `lubridate`.

```r
meeting <- ymd_hms("2011-07-01 09:00:00", tz = "Pacific/Auckland")
```

You can find what time this meeting will be in CDT time zone (America/Chicago)

```r
with_tz(meeting, "America/Chicago")
```

```
## Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'zone/tz/2018c.1.0/
## zoneinfo/Asia/Kolkata'
```

```
## [1] "2011-06-30 16:00:00 CDT"
```

you can do arithmatic with dates also for example you want to add an year to a date.

```r
leap_year(2011)
```

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

```r
ymd(20110101) + dyears(1)
```

```
## [1] "2012-01-01"
```

```r
ymd(20110101) + years(1)
```

```
## [1] "2012-01-01"
```

Functions `dyears` and `year` are identical in this respect. The difference is that `dyears` will always equal to fixed duration of 365 days. where as a `years` will take into account leap years as well. Here is an example.

```r
leap_year(2012)
```

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

```r
ymd(20120101) + dyears(1)
```

```
## [1] "2012-12-31"
```

```r
ymd(20120101) + years(1)
```

```
## [1] "2013-01-01"
```

So far we have seen seemingly well formatted character strings as input for dates which is not always the case . Dates can have months as character names or even abreviated form of three letter words. And same goes for other date components as well. You can handle that by specifying your own formats too these format builders and function `parse_date_time`.

- b : Abbreviated month name
- B : Full month name
- d : Day of the month as decimal number (01 to 31 or 0 to 31)
- H : Hours as decimal number (00 to 24 or 0 to 24). 24 hrs format
- I : Hours as decimal number (01 to 12 or 0 to 12). 12 hrs format
- j : Day of year as decimal number (001 to 366 or 1 to 366).
- m : Month as decimal number (01 to 12 or 1 to 12).
- M : Minute as decimal number (00 to 59 or 0 to 59).
- p : AM/PM indicator in the locale. Used in conjunction with I and not with H.
- S : Second as decimal number (00 to 61 or 0 to 61), allowing for up to two leap-seconds (but POSIX-compliant implementations will ignore leap seconds).
- OS :Fractional second.
- y : Year without century (00 to 99 or 0 to 99).
- Y : Year with century.

Although there are too many format builders here , you'll generally use few. we'll see example with those. you can handle hetrogenous formats as well.

```
parse_date_time("01-12-Jan","%d-%y-%b")
```

```
## [1] "2012-01-01 UTC"
```

```
parse_date_time("01-12-Jan 12:05 PM","%d-%y-%b %I:%M %p")
```

```
## [1] "2012-01-01 12:05:00 UTC"
```

```
#Function can be used seamlessely for vectors as well

x = c("09-01-01", "09-01-02", "09-01-03")
parse_date_time(x, "ymd")
```

```
## [1] "2009-01-01 UTC" "2009-01-02 UTC" "2009-01-03 UTC"
```

```
parse_date_time(x, "%y%m%d")
```

```
## [1] "2009-01-01 UTC" "2009-01-02 UTC" "2009-01-03 UTC"
```

```
parse_date_time(x, "%y %m %d")
```

```
## [1] "2009-01-01 UTC" "2009-01-02 UTC" "2009-01-03 UTC"
```

```
## ** heterogenuous formats **

x = c("09-01-01", "090102", "09-01 03", "09-01-03 12:02")
parse_date_time(x, c("%y%m%d", "%y%m%d %H%M"))
```

```
## [1] "2009-01-01 00:00:00 UTC" "2009-01-02 00:00:00 UTC"
## [3] "2009-01-03 00:00:00 UTC" "2009-01-03 12:02:00 UTC"
```

We'll conclude here. In case of any doubts regarding content of this study material, please post on QA forum in LMS.

Prepared By: Lalit Sachan

Contact: lalit.sachan@edvancer.in