R Bootcamp - Course 2: Working with Data

James Baglin

Last updated: 13 July, 2020

Overview

Summary

This course will introduce you to the management of data through R. This is a massive topic, so this course will focus on the fundamentals to help you get started. You will pick-up a range of additional and advanced data manipulation techniques throughout the remainder of the course and other courses in the program.

Learning Objectives

By the end of this course, you will have completed the following:

- Understood the concept of tidy data
- Used RStudio and R to import and export data
- Used R and the `dplyr`, `forcats` and the `lubridate` packages to perform basic data manipulations including:
  - Selecting variables
  - Assigning, reordering and labelling factors
  - Filtering and sub-setting data
  - Adding a new variable
  - Arranging or reordering your data
  - Working with dates and times

Data

This module will utilise the following data sources:

Bicycle

The `Bicycle` (data/Bicycle.csv) dataset, downloaded from the data.vic (https://www.data.vic.gov.au/data/dataset/bicycle-volumes-vicroads/resource/645eb240-b9eb-4520-bf99-ab34c2c22215) website, contains data recording cycling traffic volume recorded at 21 counter sites within Melbourne between 2005 to 2012. The dataset contains over 57,000 rows of data.

Specifically, the `Bicycle` (data/Bicycle.csv) contains the following variables:

- **Unique_ID**: Self-evident
- **NB_TRAFFIC_SURVEY**: Survey Number
- **NB_LOCATION_TRAFFIC_SURVEY**: Location survey Number
- **Sort Des**: Short Description of the location
- **DS_LOCATION**: Location Description
- **DT_ANALYSIS_SUMMARY**: Date
- **NB_YEAR**: Year data collected
- **NB_MONTH**: Month data collected
- **NB_WEEKDAY_NONHOL_QTR**: Holiday period indication
- **CT_VOLUME_AMPEAK**: Max hour in morning peak
- **CT_VOLUME_PMPEAK**: Max hour in evening peak
- **CT_VOLUME_4HOUR_OFFPEAK**: 4 hour off peak volume (12:00 to 4:00 PM)
- **CT_VOLUME_12HOUR**: 12 hour volume (7:00 AM to 7:00 PM)
- **CT_VOLUME_24HOUR**: 24 hour volume
- **DS_HOLIDAY**: Holiday description

- **NB_SEASONALITY_PERIOD**: Seasonality period indication (1 to 27)
- **NB_TYPE_PERIOD**: Seasonality period type indication (1 to 3)
- **Primary**: Primary site indication (True / False)
- **weekend**: Weekend indication (True / False)
- **Quarter**: Number quarter (1 to 4)
- **Season**: Weather season
- **Cyclying**: Season Cycling season
- **day**: Day of the week

Here is a random sample of the full dataset:

Show 10 ▼ **entries** **Search:**

	Unique_ID	NB_TRAFFIC_SURVEY	NB_LOCATION_TRAFFIC_SURVEY
1	9683	9308	1
2	15049	9311	3
3	9330	9308	1
4	19842	9288	3
5	10857	9308	1
6	2266	9306	2
7	34069	9294	3
8	36742	9295	3
9	25118	9290	3
10	21647	9288	3

Showing 1 to 10 of 100 entries

Previous 1 2 3 4 5 … 10 Next

Tidy Data

Datasets can be created from a diverse ranges of sources including manually created spreadsheets, datasets scraped from the internet, previously collected or historical data, or complex databases and data warehouses. The `Bicycle` dataset above is an example of a previously collected dataset downloaded from an Open Access data repository. Regardless of a dataset's origin, all must abide by the Tidy Data rules (Wickham 2014). As Wickham explains, tidy data allows easy manipulation, analysis and visualisation for data analysis purposes.

The basic structure of a dataset includes rows and columns. The three tidy dataset rules are as follows:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

Each variable forms a column

The `Bicycle` data contains 23 columns. With the exception of `Unique_ID`, the other columns refer to variables. For example, the `NB_YEAR` column is a time variable that tells us the year an observation was recorded. If we select only the `CT_VOLUME...` variables, we can how the `Bicycle` dataset abides by Rule #1

Show 10 ▼ **entries** **Search:**

	CT_VOLUME_AMPEAK	CT_VOLUME_PMPEAK	CT_VOLUME_4HOUR_OI
1	167	96	
2	356	277	
3	222	229	
4	35	14	
5	15	185	
6	105	98	
7	14	20	
8	105	68	
9	130	152	
10	32	345	

Showing 1 to 10 of 100 entries

Previous 1 2 3 4 5 … 10 Next

Each observation forms a row

The first row of the dataset, or the header row, includes the name of each column/variable. These names cannot contain as special characters or spaces, and their length should be as short as possible. This will reduce the amount of typing when you code!

An observation refers to the units of sampling. For example, this might be a person, a date, or a machine that comprise a "population". The sampling unit for the `bicyle` data refers to daily bike traffic volume at different locations. Location is one unit of sampling, day is the second unit of sampling. This is an example of a nested dataset, where daily traffic volumes can be nested within different locations.

Let's take a closer look at the 55,967th row of data. First we randomly sample a number between 0 and 57,003, the total number of observations in the dataset.

Here is the row of data. Scroll across to view this observation's data.

Show 1 ▼ **entries** **Search:**

	Unique_ID	NB_TRAFFIC_SURVEY	NB_LOCATION_TRAFFIC_SURVEY
1	55967	9304	3

Showing 1 to 1 of 1 entries Previous 1 Next

This is a little tricky to read, so let's transpose this two columns.

Show 23 ▼ **entries** **Search:**

	Variable	Observation
1	Unique_ID	55967
2	NB_TRAFFIC_SURVEY	9304
3	NB_LOCATION_TRAFFIC_SURVEY	3
4	Sort.Des	Scotchmans Creek Trail
5	DS_LOCATION	(BIKE PATH)SCOTCHMANS CREEK TRAIL 52M I
6	DT_ANALYSIS_SUMMARY	27/12/2009
7	NB_YEAR	2009
8	NB_MONTH	12
9	NB_WEEKDAY_NONHOL_QTR	0
10	CT_VOLUME_AMPEAK	22
11	CT_VOLUME_PMPEAK	31
12	CT_VOLUME_4HOUR_OFFPEAK	91
13	CT_VOLUME_12HOUR	239

14	CT_VOLUME_24HOUR	248
15	DS_HOLIDAY	S/HOLS
16	NB_SEASONALITY_PERIOD	23
17	NB_TYPE_PERIOD	3
18	Primary	FALSE
19	weekend	TRUE
20	Quarter	4
21	Season	Summer
22	Cyclying.Season	Cycling
23	day	Sun

Showing 1 to 23 of 23 entries

Previous 1 Next

As you can see, the 55,967th observation relates to data recorded from Scotchman's Creek Trail on the 27/12/2009. This confirms that each row with the dataset refers to an observation.

Each type of observational unit forms a table

There are two observational units in this dataset - survey location and day. You can see this in the repetition of the survey location identifiers. However, this dataset is still tidy because the table contains only information relating to daily bicycle traffic volume.

Had the dataset also contained details of a survey location's variables repeated each time to the daily observations, the dataset would have violated rule 3.

Rule three is any interesting one, because in statistics it is often violated because many standard statistical functions require it to be broken. The two main issues related to violating rule three is increased file size (not good for computation) and increased risk of inconsistencies creeping into the dataset.

Messy data

Any datasets that do not abide by these rules is defined as messy. Wickham lists five common reasons:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.

- A single observational unit is stored in multiple tables.

You can read all about them here (http://vita.had.co.nz/papers/tidy-data.pdf).

#Data Frames

R has many different types of objects capable of storing data. These include objects such as vectors, `c()`, lists, `list()`, matrices, `matrix()`, tables, `table()` and data frames, `data.frame()`. Data frames are by far the most versatile and easy to work with. Let's take a look at a quick example of how R builds a data frame. We start with toy dataset comprise of an ID, Group and Score variable. The data are in a tidy format.

Show 10 **entries** **Search:**

	ID	Group	Score
1	1	A	-0.88
2	2	A	1.11
3	3	A	0.69
4	4	A	0.92
5	5	A	0.41
6	6	B	0.24
7	7	B	-0.47
8	8	B	-0.3
9	9	B	1.22
10	10	B	-1.59

Showing 1 to 10 of 10 entries Previous 1 Next

We can create a series of vectors, `c()`, that store the data for each column.

```
x <- c(1,2,3,4,5,6,7,8,9,10)
y <- c("A","A","A","A","A","B","B","B","B","B")
z <- c(-0.88,1.11,0.69,0.92,0.41,0.24,-0.47,-0.30,1.22,-1.59)
```

Now we can build the dataset by assigning `data.frame()` to an object named `df`.

```
df <- data.frame(ID = x, Group = y, Score = z)
```

We build the header row by assigning variables names as `ID =` or `Group =`. This is what the `data.frame()` will look like:

```
## # A tibble: 10 x 3
##       ID Group Score
##    <dbl> <fct> <dbl>
##  1     1 A     -0.88
##  2     2 A      1.11
##  3     3 A      0.69
##  4     4 A      0.92
##  5     5 A      0.41
##  6     6 B      0.24
##  7     7 B     -0.47
##  8     8 B     -0.3
##  9     9 B      1.22
## 10    10 B     -1.59
```

Data frames are easy to work with and also recognise matrix commands

```
#Select a variable
df$Score
```

```
##  [1] -0.88  1.11  0.69  0.92  0.41  0.24 -0.47 -0.30  1.22 -1.59
```

```
#Select a variable using matrix code
df[,2]
```

```
## # A tibble: 10 x 1
##    Group
##    <fct>
##  1 A
##  2 A
##  3 A
##  4 A
##  5 A
##  6 B
##  7 B
##  8 B
##  9 B
## 10 B
```

```
#Select a specific row
df[ID == 5,]
```

```
## # A tibble: 1 x 3
##      ID Group Score
##   <dbl> <fct> <dbl>
## 1     5 A      0.41
```

Data imported into R defaults to data.frames. Data frames can also be readily converted to other object types. The best data manipulation packages also assume you are using data frames.

Importing and Exporting Data

The following slides will take your through the process of importing a `.csv` dataset into R using RStudio. RStudio also allows you to open data stored in a wide range of file types including Excel, SPSS, SAS and Stata. All datasets must follow the Tidy Data rules.

You can also import data directly using code. The code depends on the type of dataset being imported. To import a `.csv` file we can use the `read.csv()` function. You may need to provide a direct path to your `.csv` file depending on where it is located. For example…

```
Bicycle <- read.csv("C:/OneDrive/Data Repository/Bicycle.csv")
```

Note the use of forward slashes. If you are working from an RStudio Project and the dataset is located in the project folder, the code changes to…

```
Bicycle <- read.csv("Bicycle.csv")
```

This is one of the major advantages of using projects. When you move to another computer, the relative path will still work.

If the file was tab delimited, `txt`, we can change to the more general `read.table()` function. We have to instruct R how the dataset is delimited using `sep =` and also to treat the first row of data as the header row, `header = TRUE`.

```
Bicycle <- read.table("Bicycle.txt", header = TRUE, sep = "\t")
```

If the data was in Excel format, we can use the `readxl` package.

```
library(readxl)
Bicycle <- read_excel("Bicycle.xls")
```

Once again, the dataset must be in a Tidy Data format for these importation methods to work. If these methods fail, re-check your data.

If the importation is successful, you can use the `View()` function or click on the dataset object name

Sometimes you need to get data out of R. You can use the `write.table` function for this purpose.

```
write.table(df, file = "experiment.csv", sep = ",", col.names = TRUE, ro
  w.names = FALSE)
```

`col.names` ensures a header row is included, `row.names` suppressed the printing of row numbers (you don't tend to need this) and `sep` determines the file type, in this example comma-delimited. The `file` argument names and locates where the file will be saved.

There are also shortcut functions for common data file types:

```
write.csv(df, file = "experiment.csv")
```

Factors

Assigning Factors

Categorical variables (qualitative, nominal and ordinal) are referred to as Factors in R. Factors are comprised of levels. For example, the factor `DS_LOCATION` has 34 levels corresponding to 34 different survey sites.

```
Bicycle$DS_LOCATION %>% levels()
```

```
##  [1] "(BIKE LANE)BRIGHTON RD N BD 50M S OF MOZART ST"
##  [2] "(BIKE LANE)BRIGHTON RD S BD 50M N OF DICKENS ST"
##  [3] "(BIKE LANE)FLEMINGTON RD NW BD 10M SE OF DRYBURGH ST"
##  [4] "(BIKE LANE)FLEMINGTON RD SE BD 25M NW OF ABBOTSFORD ST"
##  [5] "(BIKE LANE)ROYAL PDE N BD 10M N OF GATEHOUSE ST"
##  [6] "(BIKE LANE)ROYAL PDE S BD 20M S OF GATEHOUSE ST"
##  [7] "(BIKE LANE)ST. KILDA RD N BD 25M N OF CONVENTRY ST"
##  [8] "(BIKE LANE)ST. KILDA RD S BD 30M S OF ANZAC AVE"
##  [9] "(BIKE PATH)ANN TRAIL NO:1 2WAY 104M SOUTH OF OF WHITEHORSE RD"
## [10] "(BIKE PATH)ANN TRAIL NO:2 2WAY 300M EAST OF PRINCESS ST"
## [11] "(BIKE PATH)BAY TRAIL 2WAY 100M N BLESSINGTON ST OPP NO 20 MARINE
PDE"
## [12] "(BIKE PATH)CANNING ST 2WAY PRINCESS ST OUTSIDE DAN O'CONNELL"
## [13] "(BIKE PATH)CAPITAL CITY TRAIL 2WAY 25M W OF BOWEN CRES"
## [14] "(BIKE PATH)CAPITAL CITY TRAIL 2WAY FOOTSCRAY RD SE OF EXIT RAMP
FROM CITY LINK"
## [15] "(BIKE PATH)GARDINERS CREEK TRAIL 2WAY 66M W OF CITYLINK OVERPAS
S"
## [16] "(BIKE PATH)GARDNERS CREEK TRAIL NO.2 2WAY ADJ ESTELLA ST"
## [17] "(BIKE PATH)KOONUNG TRAIL 2WAY 44M NE OF CLIFTON ST"
## [18] "(BIKE PATH)MAIN YARRA TRAIL NO:1 2WAY ALONG YARRA BVD 66M W OF O
F C'LINK OPASS"
## [19] "(BIKE PATH)NORTH BANK 2WAY 14M E OF MORRELL BRIDGE OF ADJACENT T
O PUNT RD O"
## [20] "(BIKE PATH)NORTH BANK 2WAY 75M W OF MORELL BRIDGE ADJACENT TO PU
NT RD OVERPASS"
## [21] "(BIKE PATH)SCOTCHMANS CREEK TRAIL 52M E OF 61 SMYTH ST"
## [22] "(BIKE PATH)SOUTH BANK 2WAY UNDER PUNT RD BRIDGE"
## [23] "(BIKE PATH)ST. GEORGES RD 2WAY 28M S OF SUMNER AV"
## [24] "(BIKE PATH)ST. GEORGES RD NO.2 2W N OF BELL ST"
## [25] "(BIKE PATH)ST. GEORGES ST 2WAY 50M S OF HAWTHORN RD TEST SITE"
## [26] "(BIKE PATH)TRAM 109 TRAIL 2WAY 10M NE OF ACCESS PATH CNR WOODGAT
E & BOUNDARY STS"
## [27] "(BIKE PATH)UPFIELD RAILWAY LIN 2WAY 10M S OF PARK ST"
## [28] "(BIKELANE) ALBERT ST EB 50M E OF MORRISON PL"
## [29] "(BIKELANE) ALBERT ST WB 50M W OF LANSDOWN ST"
## [30] "(BIKEPATH) FEDERATION TRAIL 170M SE OF PRINCESS HWY BTW CYPRESS
AV & CONIFER AV"
## [31] "(BIKEPATH) MERRIE CREEK TRAIL 2WAY S OF MORELAND RD"
## [32] "(BIKEPATH) MORELAND ST PATH 2WAY 50 M N OF PARKER ST"
## [33] "(BIKEPATH) NAPIER ST PATH 2WAY 100M N OF GREEVES ST"
## [34] "(BIKEPATH) PHILLIP ISLAND RD PATH BTW BUNVEGAN CR & GLEN ST"
```

Sometimes R imports data without knowing what it really means. For example, `NB_TRAFFIC_SURVEY` has been imported as an integer, `int`.

```
Bicycle$NB_TRAFFIC_SURVEY %>% class()
```

```
## [1] "integer"
```

However, this variable corresponds to a survey identifier, or nominal variable. The nominal scale of this variable is not meaningful, in the same way as a credit card number. We cannot meaningfully apply mathematical operations. We can tell R to treat this variable as a factor using the following code:

```
Bicycle$NB_TRAFFIC_SURVEY <- Bicycle$NB_TRAFFIC_SURVEY %>% as.factor()
Bicycle$NB_TRAFFIC_SURVEY %>% class
```

```
## [1] "factor"
```

The first part of the code selects the Bicycle object and we use the `$` sign to signal the selection of a variable, in this case `NB_Traffic_Survey`. We then instruct R to copy over this variable using the same values from `Bicycle$NB_TRAFFIC_SURVEY` columns, but as a factor, using the `as.factor()` function. If you're successful, the `NB_TRAFFIC_SURVEY` variable will be updated as a Factor with 29 levels. Now R will treat this variable as a numeric label and not a meaningful integer.

R will convert factors to numeric levels based on numeric/alphabetical order. Let's consider the `day` variable:

```
levels(Bicycle$day)
```

```
## [1] "Fri" "Mon" "Sat" "Sun" "Thu" "Tue" "Wed"
```

So, R will treat Fri = 1, Mon = 2, Sat = 3 etc. This isn't good and this is why you need to pay careful attention to factors in your dataset.

Re-order Factor

We can reorder factor levels using the `factor()` function. Here's how:

```
Bicycle$day <- Bicycle$day %>%  factor(
  levels=c('Sun','Mon','Tue','Wed','Thu','Fri','Sat'),
  ordered=TRUE)
Bicycle$day %>% levels
```

```
## [1] "Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"
```

So, Sun = 1, Mon = 2, etc. Nice work. The `ordered = TRUE` option ensures the ordering of days is meaningful.

We can also use the extremely useful functions for dealing with factors from the `forcats` package. For example, if we wanted Monday first:

```
library(forcats)
Bicycle$day <- Bicycle$day %>% fct_relevel('Mon','Tue','Wed','Thu','Fri',
  'Sat','Sun')
Bicycle$day %>% levels
```

```
## [1] "Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"
```

Assigning Labels

Sometimes we need to assign different labels to factors. Let's look at the weekend variable. This is a logical/binary variable where `TRUE` indicates a Saturday or Sunday observation, otherwise for weekdays it's `FALSE`. Let's change this variable to a more descriptive version using base code.

```
Bicycle$weekend <- Bicycle$weekend %>% as.factor
Bicycle$weekend %>% levels
```

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

```
Bicycle$weekend <- Bicycle$weekend %>% factor(levels = c("FALSE","TRUE"),
                                              labels=c("Weekday","Weeken
  d"))
Bicycle$weekend %>% levels
```

```
## [1] "Weekday" "Weekend"
```

Had we used the `forcats` package instead (don't run this code if you ran the previous):

```
Bicycle$weekend <- Bicycle$weekend %>% as.factor
Bicycle$weekend <- Bicycle$weekend %>% fct_recode("Weekend" = "TRUE",
                                                  "Weekday"  = "FALSE")
```

You can see how `fct_recode` does not require levels to be specified.

Converting Numeric Values to Factors

Sometimes we need to store numeric values as factors. Let's look at the `Quarter` variable. This variable records the quarter of year as 1, 2, 3, or 4. Let's convert this variable to a factor and label the factor accordingly.

```
Bicycle$Quarter <- Bicycle$Quarter %>% factor(levels=c(1,2,3,4),
    labels=c("1st Quarter","2nd Quarter","3rd Quarter", "4th Quarter"),
    ordered = TRUE)
```

Levels refer to the numeric values defining each quarter. Labels define a descriptive label for each level. Order ensures R treats the factor as an ordinal variable.

Filtering and Sub-setting Data

We will use the powerful and intuitive functions from the `dplyr` package to demonstrate sub-setting and filtering data. This is one of the most common data manipulation techniques that you need to master.

For example, you might want to select and analyse only the data pertaining to summer. We can use the `filter()` function from the `dplyr` package for this purpose:

```
library(dplyr)
Bicycle_Summer <- Bicycle %>% filter(Season == "Summer")
Bicycle_Summer$Season %>% summary
```

```
## Autumn Spring Summer Winter
##      0      0  13757      0
```

This code will create a new data frame object, called `Bicycle_Summer` by selecting only the cases where season is equal to "Summer". Note the use of the logical operator "==", which means "equal to".

We can quickly build more complete filters by joining logical operators. This time, let's select observations from summer and spring:

```
Bicycle_Summer_Spring <- Bicycle %>% filter(Season=="Summer" | Season ==
  "Spring")

# Check

Bicycle_Summer_Spring$Season %>% summary
```

```
## Autumn Spring Summer Winter
##      0  15254  13757      0
```

Note the use of the "|" logical operator which means "or".

We can also add more complex filters by referring to other variables. The following code selects all observations from Summer or Spring after the Year 2009.

```
Bicycle_Summer_Spring_2009 <- Bicycle %>% filter((Season=="Summer" | Seas
  on =="Spring")
                                                 & NB_YEAR >= 2009)
```

Here is a frequency table of the original and filtered datasets

```
table(Bicycle$Season, Bicycle$NB_YEAR)
```

```
##
##          2005 2006 2007 2008 2009 2010 2011 2012 2013
##   Autumn   14 1028 1480 1566 1781 1557 2457 3317    0
##   Spring   40 1335 1508 1815 1597 1671 3697 3591    0
##   Summer  238 1244 1472 1578 1669 1671 2574 3059  252
##   Winter   20 1020 1518 1868 1643 1765 3592 3366    0
```

```
# Check

table(Bicycle_Summer_Spring_2009$Season, Bicycle_Summer_Spring_2009$NB_YE
  AR)
```

```
##
##          2009 2010 2011 2012 2013
##   Autumn    0    0    0    0    0
##   Spring 1597 1671 3697 3591    0
##   Summer 1669 1671 2574 3059  252
##   Winter    0    0    0    0    0
```

Always confirm your filter by checking the new data frame object.

Adding New Variables

You can create new variables in a dataset by recoding or manipulating existing variables. For example, if we did not have the `weekend` variable, we could use the `day` variable to create it. There are many ways to do this. We will take a look at using `fct_recode` function to do this quickly. To add a new variable, we assign a new variable name to the `data.frame`:

```
Bicycle$New_weekday_variable <- Bicycle$day %>% fct_recode("weekday" = "M
  on",
                                                                  "weekday" = "T
  ue",
                                                                  "weekday" = "W
  ed",
                                                                  "weekday" = "T
  hu",
                                                                  "weekday" = "F
  ri",
                                                                  "weekend" = "S
  at",
                                                                  "weekend" = "S
  un")

# Check

table(Bicycle$New_weekday_variable, Bicycle$day)
```

```
##
##            Mon  Tue  Wed  Thu  Fri  Sat  Sun
##   weekday 8109 8125 8131 8087 8098    0    0
##   weekend    0    0    0    0    0 8267 8186
```

You can also add new variables as transformation of existing variables. For example we can create a total "peak" proportion tragic variable by adding the `CT_VOLUME_AMPEAK` and `CT_VOLUME_PM_PEAK` values and dividing by `CT_VOLUME_24HOUR`. Or…

$$\text{Peak} = \frac{\text{AM Peak} + \text{PM Peak}}{24 \text{ Total}}$$

This variable will tell us the proportion of daily traffic that occurred during morning and evening peak times.

We can use `mutate()` function from the `dplyr` package.

```
Bicycle <- Bicycle %>% mutate(peak = (CT_VOLUME_AMPEAK + CT_VOLUME_PMPEA
  K)/CT_VOLUME_24HOUR)
```

Why use `mutate`? If we didn't this is how the code would look.

```
Bicycle$peak <- (Bicycle$CT_VOLUME_AMPEAK + Bicycle$CT_VOLUME_PMPEAK)/Bic
  ycle$CT_VOLUME_24HOUR
```

The constant reference to the `Bicycle` data object seems redundant.

You can read more about the advantages of using the `dplyr` package here (https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html).

Selecing Variables

Sometimes we want to narrow the focus of our dataset by considering only a subset of variables. For example, in the `bicycle` dataset we might need to extract only the location, date and 24 hour traffic volume variables. We can achieve this using the `select()` function from the `dplyr` package.

```
Bicycle_volume <- Bicycle %>% dplyr::select(Sort.Des, DT_ANALYSIS_SUMMAR
  Y, CT_VOLUME_24HOUR)
Bicycle_volume %>% head()
```

```
## # A tibble: 6 x 3
##   Sort.Des                    DT_ANALYSIS_SUMMARY CT_VOLUME_24HOUR
##   <fct>                       <fct>                          <int>
## 1 St Georges St Hawthorn      09/06/2007                       480
## 2 Flemington Rd NW Bound Lane 17/03/2008                       654
## 3 Flemington Rd NW Bound Lane 19/03/2008                       794
## 4 Flemington Rd NW Bound Lane 20/03/2008                       732
## 5 Flemington Rd NW Bound Lane 21/03/2008                       221
## 6 Flemington Rd NW Bound Lane 22/03/2008                       271
```

Ordering Datasets

Ordering a dataset facilitate easy searching and be necessary for certain statistical functions. To order a dataset we can use the `arrange()` function from the `dplyr` package. Let's order the bicycle dataset by `Sort.Des`, `NB_YEAR` and `NB_MONTH`.

```
Bicycle_sorted <- Bicycle %>% arrange(Sort.Des,NB_YEAR,NB_MONTH)
Bicycle_sorted %>% head()
```

```
## # A tibble: 6 x 25
##   Unique_ID NB_TRAFFIC_SURV~ NB_LOCATION_TRA~ Sort.Des DS_LOCATION
##       <int> <fct>                       <int> <fct>    <fct>
## 1     17386 9316                            1 Albert ~ (BIKELANE)~
## 2     17387 9316                            1 Albert ~ (BIKELANE)~
## 3     17388 9316                            1 Albert ~ (BIKELANE)~
## 4     17389 9316                            1 Albert ~ (BIKELANE)~
## 5     17390 9316                            1 Albert ~ (BIKELANE)~
## 6     17391 9316                            1 Albert ~ (BIKELANE)~
## # ... with 20 more variables: DT_ANALYSIS_SUMMARY <fct>, NB_YEAR <int
>,
## #   NB_MONTH <int>, NB_WEEKDAY_NONHOL_QTR <int>, CT_VOLUME_AMPEAK <int
>,
## #   CT_VOLUME_PMPEAK <int>, CT_VOLUME_4HOUR_OFFPEAK <int>,
## #   CT_VOLUME_12HOUR <int>, CT_VOLUME_24HOUR <int>, DS_HOLIDAY <fct>,
## #   NB_SEASONALITY_PERIOD <int>, NB_TYPE_PERIOD <int>, Primary <lgl>,
## #   weekend <fct>, Quarter <ord>, Season <fct>, Cyclying.Season <fct>,
## #   day <ord>, New_weekday_variable <ord>, peak <dbl>
```

Working with Dates and Times

Working with dates and times in R can be a little tricky, however, with the use of some useful packages and functions things can remain relatively straight forward. Let’s take a look at the `DT_ANALYSIS_SUMMARY` variable.

```
Bicycle$DT_ANALYSIS_SUMMARY %>% class
```

```
## [1] "factor"
```

This is wrong because the values refer to dates:

```
Bicycle$DT_ANALYSIS_SUMMARY %>% head
```

```
## [1] 09/06/2007 17/03/2008 19/03/2008 20/03/2008 21/03/2008 22/03/2008
## 2637 Levels: 01/01/2006 01/01/2007 01/01/2008 01/01/2009 ... Thursday,
24 February 2011
```

The dates are in dd/mm/yyyy. The `lubridate` package provides a powerful set of functions for working with dates. Let’s fix the `DT_ANALYSIS_SUMMARY` variable.

```
library(lubridate)
Bicycle$DT_ANALYSIS_SUMMARY <- Bicycle$DT_ANALYSIS_SUMMARY %>% dmy
Bicycle$DT_ANALYSIS_SUMMARY %>%  class
```

```
## [1] "Date"
```

Excellent. As you can see, the `dmy()` function tells R the format of the date according to dd/mm/yyyy. You can use other combination like `ymd`, `mdy`, etc.

You can also apply other useful functions to extract important information once the data class has been applied. Check out the following:

```
# Extract day or week

Bicycle$DT_ANALYSIS_SUMMARY %>% wday(label = TRUE) %>%  head
```

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

```
# Extract month

Bicycle$DT_ANALYSIS_SUMMARY %>% month(label = TRUE) %>%  head
```

```
## [1] Jun Mar Mar Mar Mar Mar
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... <
Dec
```

```
# Extract year

Bicycle$DT_ANALYSIS_SUMMARY %>% year %>%  head
```

```
## [1] 2007 2008 2008 2008 2008 2008
```

`lubridate` is also excellent at handling time. For example, let's consider the following top 5, marathon times in a common hh:mm:ss format.

```
times <- c("02:02:57","02:03:03","02:03:05","02:03:13","02:03:13")
times %>% class
```

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

Now we can convert them to time format:

```
times <- times %>% hms
times %>% class
```

```
## [1] "Period"
## attr(,"package")
## [1] "lubridate"
```

We can do a wide range of useful transformations of time once we have the data in a time format.

```
# Extract seconds
times %>% second
```

```
## [1] 57  3  5 13 13
```

```
# Extract minutes

times %>% minute
```

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

```
# Convert to seconds

times %>% seconds
```

```
## [1] "7377S" "7383S" "7385S" "7393S" "7393S"
```

```
# Calculate difference between top two times in seconds

times[2] %>% seconds - times[1] %>% seconds
```

```
## [1] "6S"
```

Close!

References

Wickham, H. 2014. "Tidy data." *Journal of Statistical Software* 59 (10). https://doi.org/10.18637/jss.v059.i10 (https://doi.org/10.18637/jss.v059.i10).
