A quick tour of R

Abhijit Dasgupta, PhD

The bare essentials

- R is a **language**
- Everything in R is an object with a name
- Things are nouns
  - Nouns in R are *data objects*, like scalars, matrices, data.frames/tibbles, strings, vectors
- Nouns are acted upon by verbs
  - Verbs in R are *functions*, like `mean(x)`, `nrow(d)`, `dim(d)`, `ggplot` and so on
- You can modify verbs with adverbs
  - Adverbs in R are *function options*, like `mean(x, na.rm=T)`, `geom_point(color='green')`

The bare essentials

- If you want to save and re-use an object (noun or verb), you have to name it
  - This is done with the `<-` operator, e.g.
    - `beaches <- read_csv('data/sydneybeaches3.csv')` (dataset)
    - `mn <- mean(x, na.rm=T)` (a number)
    - `my_theme <- function() {theme_bw() + theme(axis.title=element_text(size=14))}` stores the function, which you'll call as `my_theme()`

> You can see the objects you have created either by typing `ls()` in the console, or looking in the Environment pane
>
> Note, built-in objects don't show up in the Environment pane or using `ls()`

The bare essentials

Naming rules

1. Identifiers can be a combination of letters, digits, period (.) and underscore (_).
2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit.
3. Reserved words in R cannot be used as identifiers.

Naming conventions

1. Make names expressive
2. Write multi-word names in CamelCase or snake_case

The bare essentials

Allowing non-conventional names

If a name doesn't follow the rules *per se*, it can still be used by quoting them using backticks.

So the following names are valid, as long as backticks are present.

```
`Pelvic girdle`
`1`
`Sample 03/23/2019`
```

This is rather difficult to type on a regular basis, and can lead to syntactical errors

---

The **janitor** package has a `clean_names` function that can fix the names of columns when you import datasets, transforming them into snake case by default.

The bare essentials (brackets)

`[]` are used for extracting elements from arrays, matrices, data frames.

- `x[3]` is the 3rd element of an array `x`
- `d[1,3]` is the element in the 1st row and 3rd column of a matrix/data frame `d`
- `d[2,]` is the entire first **row** of a matrix/data frame `d`
- `d[,4]` is the entire 4th **column** of a matrix/data frame `d`
- `d[,'gender']` is the column named *gender* in a *data frame* `d`

For a list object (we'll meet them in a few slides) you use a double bracket to extract elements

- `lst[[3]]` is the third element of the list `lst`
- `lst[['apple']]` is the element named *apple* in a named list `lst`

The bare essentials (brackets)

There are three kinds of brackets in R

`()` are used for specifying arguments to functions

- `mean(x)` gives the mean of an array of numbers `x`
- `summary(d)` gives a summary representation of a data frame `d`
- There are several functions used to make the following visualization

```
ggplot(beaches, aes(x = temperature, y = rainfall)) +
  geom_point() +
  geom_smooth() +
  theme_classic()
```

The bare essentials (brackets)

There are three kinds of brackets in R

{} are used to contain groups of commands/statements

A conditional statement

```
if (age < 18){
   person <- 'Minor'
} else if (age > 65) {
  person <- 'Senior'
} else {
  person <- 'Adult'
}
```

A function definition

```
my_mean <- function(x, na.rm = T){
  if(na.rm){
    x <- x[!is.na(x)]
  }
  s <- sum(x)
  n <- length(x)
  mn <- s / n # There is a built-in function mean, sc
  return(mn)
}
```

See reading and supplementary materials for more details about these constructs

Data structures

A **scalar** (data taking a single value)

- `29`
- `"cherry"`
- `TRUE`

Data structures

A **scalar** (data taking a single value)

- `29` : *numeric*
- `"cherry"`: *character*
- `TRUE`: *logical*

These are different data *types*, which determine how they are stored in memory. You may also see *integer* and *double*, which are both covered under *numeric*.

You can check an object's type using `class`: `class(29)` (resulting in `numeric`)
You can check if an object is of a particular type: `is.character(29)` (resulting in `FALSE`)
You can transform an object to a different type, if allowed: `as.numeric("29")` (resulting in `29`)

The `is.___` and `as.___` functions are extremely useful for data cleaning and transformations

You typically use the variable names rather than the actual values for these functions

Data structures

Vectors/Arrays

These are constructed using the `c()` function (for *concatenate*).

```
c(1,2,5,6,7,8)
```

```
#>  [1] 1 2 5 6 7 8
```

```
c('apple','berry','melon','citrus')
```

```
#>  [1] "apple"  "berry"  "melon"  "citrus"
```

Vectors must all contain objects of the same type. Can't mix and match

If you check the `class` of an array, it will give the common data type of the elements of the array

Data structures

Matrices (2-d arrays)

These are typically built from vectors

```
x <- c(1,2,4,5,6,7)
y <- 10:16  # Shortcut for c(10,11,12,13,14,15,16)
```

```
cbind(x, y) # Vectors as columns
```

```
#>       x  y
#>  [1,] 1 10
#>  [2,] 2 11
#>  [3,] 4 12
#>  [4,] 5 13
#>  [5,] 6 14
#>  [6,] 7 15
#>  [7,] 1 16
```

```
rbind(x, y) # Vectors as rows
```

```
#>    [,1] [,2] [,3] [,4] [,5] [,6] [,7]
#>  x    1    2    4    5    6    7    1
#>  y   10   11   12   13   14   15   16
```

Data structures

Matrices (2-d arrays)

You can also build matrices directly using the `matrix` function.

```
matrix(seq(1, 6), nrow = 3) # Automatically determines number of columns
```

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

Play around with different ways of creating vectors and matrices

Data structures

Lists

Lists are basically buckets or containers. Each element of a list can be anything, even other lists

```
my_list <- list('a', c(2,3,5,6), head(ggplot2::mpg))
my_list
```

This has a scalar, a vector and a data set

Since lists can be composed of heterogeneous objects, the `class` of a list is `list`.

```
#> [[1]]
#> [1] "a"
#>
#> [[2]]
#> [1] 2 3 5 6
#>
#> [[3]]
#> # A tibble: 6 x 11
#>   manufacturer model displ  year   cyl trans
#>   <chr>        <chr> <dbl> <int> <int> <chr>
#> 1 audi         a4      1.8  1999     4 auto(l5)
#> 2 audi         a4      1.8  1999     4 manual(m5)
#> 3 audi         a4      2    2008     4 manual(m6)
#> 4 audi         a4      2    2008     4 auto(av)
#> 5 audi         a4      2.8  1999     6 auto(l5)
#> 6 audi         a4      2.8  1999     6 manual(m5)
```

Data structures

Lists

You can create named lists, where every element has a name

```
my_list2 <- list('scalar' = 'a', 'vector' = c(2,3,5,6), 'data' = head(ggplot2::mpg))
my_list2[['vector']]
```

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

```
my_list2[['data']]
```

```
#>  # A tibble: 6 x 11
#>    manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
#>    <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
#>  1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa…
#>  2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa…
#>  3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa…
#>  4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa…
#>  5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compa…
#>  6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p     compa…
```

Data structures

data.frame/tibble

This is the typical container for tabular data

- must be rectangular
- each column can be of a different type
- elements within each column have to be of the same type

The data frame is probably the single most important structure in R for data analysis. It allows you to keep different kinds of data for each observation together, regardless of type. Many other scripting systems couldn't keep heterogeneous data together, which made practical data analyses very difficult.

This kind of structure is now present in almost every serious data analytic platform, including Python, Julia, SAS, SPSS. Matlab remains a significant exception.

Data structures

data.frame/tibble

```
# use the import function from the package rio
beaches <- rio::import('data/sydneybeaches3.csv')
class(beaches)
```

```
#>  [1] "data.frame"
```

```
dim(beaches)
```

```
#>  [1] 344  12
```

```
head(beaches)
```

```
#>          date year month day season rainfall temperature enterococci day_num
#>  1 2013-01-02 2013     1   2      1      0.0        23.4          6.7       2
#>  2 2013-01-06 2013     1   6      1      0.0        30.3          2.0       6
#>  3 2013-01-12 2013     1  12      1      0.0        31.4         69.1      12
#>  4 2013-01-18 2013     1  18      1      0.0        46.4          9.0      18
#>  5 2013-01-24 2013     1  24      1      0.0        27.5         33.9      24
#>  6 2013-01-30 2013     1  30      1      0.6        26.6         26.5      30
#>    month_num month_name season_name
```

Data structures

data.frame/tibble

```
library(tidyverse) # Activate the tidyverse package
beaches_t <- as_tibble(beaches)
class(beaches_t)
```

```
#>  [1] "tbl_df"     "tbl"        "data.frame"
```

```
beaches_t
```

```
#>  # A tibble: 344 x 12
#>     date        year month   day season rainfall temperature enterococci day_num
#>     <date>     <int> <int> <int>  <int>    <dbl>       <dbl>       <dbl>   <int>
#>   1 2013-01-02  2013     1     2      1      0          23.4         6.7       2
#>   2 2013-01-06  2013     1     6      1      0          30.3         2         6
#>   3 2013-01-12  2013     1    12      1      0          31.4        69.1      12
#>   4 2013-01-18  2013     1    18      1      0          46.4         9        18
#>   5 2013-01-24  2013     1    24      1      0          27.5        33.9      24
#>   6 2013-01-30  2013     1    30      1      0.6        26.6        26.5      30
#>   7 2013-02-05  2013     2     5      1      0.1        25.7        66.9      36
#>   8 2013-02-11  2013     2    11      1      8          22.2       118.       42
#>   9 2013-02-17  2013     2    17      1     13.6        26.3        75        48
#>  10 2013-02-23  2013     2    23      1      7.2        24.8       311.       54
#>  # … with 334 more rows, and 3 more variables: month_num <int>,
```

Data structures

data.frame/tibble

We can take a quick look at the data types of each column

```
str(beaches)
```

```
#>  'data.frame':    344 obs. of  12 variables:
#>   $ date       : IDate, format: "2013-01-02" "2013
#>   $ year       : int  2013 2013 2013 2013 2013 201
#>   $ month      : int  1 1 1 1 1 1 2 2 2 2 ...
#>   $ day        : int  2 6 12 18 24 30 5 11 17 23 .
#>   $ season     : int  1 1 1 1 1 1 1 1 1 1 ...
#>   $ rainfall   : num  0 0 0 0 0 0.6 0.1 8 13.6 7.2
#>   $ temperature: num  23.4 30.3 31.4 46.4 27.5 26.
#>   $ enterococci: num  6.7 2 69.1 9 33.9 ...
#>   $ day_num    : int  2 6 12 18 24 30 36 42 48 54
#>   $ month_num  : int  1 1 1 1 1 1 2 2 2 2 ...
#>   $ month_name : chr  "January" "January" "January
#>   $ season_name: chr  "Summer" "Summer" "Summer" '
```

```
glimpse(beaches)
```

```
#>  Rows: 344
#>  Columns: 12
#>  $ date        <date> 2013-01-02, 2013-01-06, 2013
#>  $ year        <int> 2013, 2013, 2013, 2013, 2013,
#>  $ month       <int> 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
#>  $ day         <int> 2, 6, 12, 18, 24, 30, 5, 11,
#>  $ season      <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
#>  $ rainfall    <dbl> 0.0, 0.0, 0.0, 0.0, 0.0, 0.6,
#>  $ temperature <dbl> 23.4, 30.3, 31.4, 46.4, 27.5,
#>  $ enterococci <dbl> 6.700000, 2.000000, 69.100000
#>  $ day_num     <int> 2, 6, 12, 18, 24, 30, 36, 42,
#>  $ month_num   <int> 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
#>  $ month_name  <chr> "January", "January", "Januar
#>  $ season_name <chr> "Summer", "Summer", "Summer",
```

Data structures

data.frame/tibble

Extracting columns from a data frame:

1. `beaches$temperature` (works during exploration or when only single column needed)
2. `beaches['temperature']` (*This is my preferred method*)
3. `beaches[['temperature']]`
4. `beaches[,7]`
5. `beaches[[7]]`

Packages in R

R is a modular environment with some base functionality that is augmented by **packages** (think of them as modules)

- Packages can contain *functions* and *data*
- There are over 15K packages on CRAN, the Comprehensive R Archive Network
- There are over 1600 packages on Bioconductor, the main repository for bioinformatics resources
    - Analytic, Annotation, Experimental data and Workflow packages

Finding packages

1) CRAN Task views
2) Bioconductor BiocViews
3) GitHub (open source collaboration and version control environment)

Installing packages

From CRAN

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

From Bioconductor

```
install.packages("BiocManager") # do once
BioCManager::install('limma')
```

From GitHub

```
install.packages('remotes') # do once
remotes::install_github("rstudio/rmarkdown")
# usual format is username/packagename
```

> GitHub often hosts development version of packages published on CRAN or Bioconductor
>
> Both CRAN and Bioconductor have stringent checks to make sure packages can run properly, with no obvious program flaws. There are typically no guarantees about analytic or theoretical correctness, but most packages have been crowd-validated and there are several reliable developer groups including RStudio

Using packages

You have to first "activate" the package in your current working session using the `library` function.

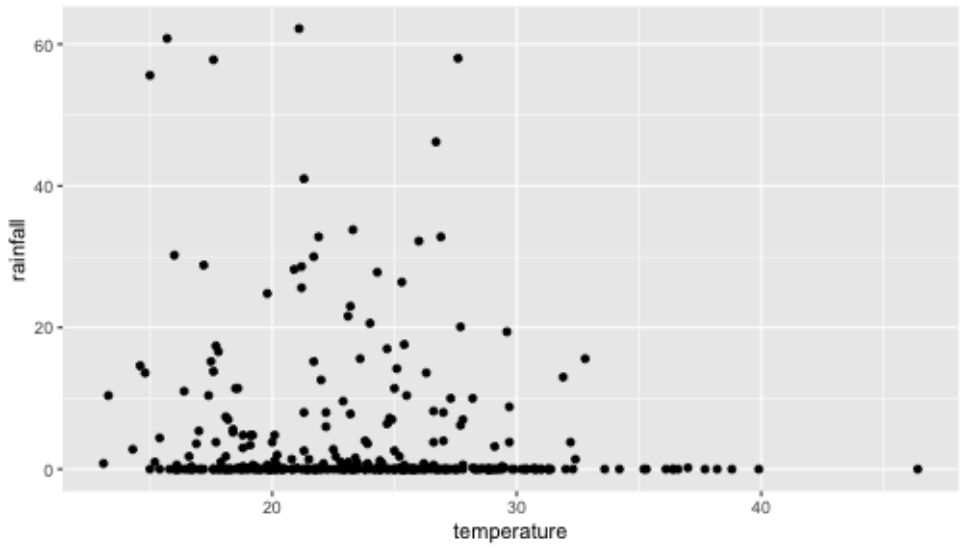
```
ggplot(beaches, aes(temperature, rainfall)) +
  geom_point()
```

```
#>  Error in ggplot(beaches, aes(temperature, rainfal
```

The first thing you look for in this error message is which package isn't loaded. Then either load it, or install it, as the case may be

Recall, you have to
install packages once *per computer*, but
load packages once *per session*

```
library(ggplot2) # or library(tidyverse)
ggplot(beaches, aes(temperature, rainfall)) +
  geom_point()
```



Tidying data using the tidyverse

What is the "Tidyverse"?

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures

What is the "Tidyverse"?

A set of R packages that:

- help make data more computer-friendly
- while making your code more human-friendly

- Most of these packages are (co-)written by Dr. Hadley Wickham, who has rockstar status in the R world
- They are supported by the company RStudio

The tidyverse.org site and the R4DS book are the definitive sources for tidyverse information.

The packages are united in a common philosophy of how data analysis should be done.

Tidying data

Tidy data

Tidy datasets are all alike,
but every messy data is messy in its own way

Tidy data

Tidy data is a **computer-friendly** format based on the following characteristics:

- Each row is one observation
- Each column is one variable
- Each set of observational unit forms a table

All other forms of data can be considered **messy data**.

Let us count the ways

There are many ways data can be messy. An incomplete list....

- Column headers are values, not variables
- Multiple variables are stored in a single column
- Variables are stored in both rows and columns
- Multiple types of observational units are saved in the same table
- A single observational unit is stored in multiple tables

Ways to have messy (i.e. not tidy) data

1. Column headers contain values

Country	< $10K	$10-20K	$20-50K	$50-100K	> $100K
India	40	25	25	9	1
USA	20	20	20	30	10

Ways to have messy (i.e. not tidy) data

Column headers contain values

Country	Income	Percentage
India	< $10K	40
USA	< $10K	20

This is a case of reshaping or melting

Ways to have messy (i.e. not tidy) data

Multiple variables in one column

Country	Year	M_0-14	F_0-14	M_ 15-60	F_15-60	M_60+	F_60+
UK	2010						
UK	2011						

Separating columns into different variables

Country	Year	Gender	Age	Count

Tidying data

The typical steps are

- Transforming data from wide to tall (`pivot_longer`) and from tall to wide (`pivot_wider`)
- Separating columns into different columns (`separate`)
- Putting columns together into new variables (`unite`)

> The functions `pivot_longer` and `pivot_wider` supercede the older functions `gather` and `spread`, which I have used in previous iterations of this class. However, if you are familiar with `gather` and `spread`, they aren't gone and can still be used in the current **tidyr** package.

Cleaning data

Some actions on data

- Creating new variables (`mutate`)
- Choose some columns (`select`)
- Selecting rows based on some criteria (`filter`)
- Sort data based on some variables (`arrange`)
- Reduce multiple values to a single summary (`summarize`/`summarise`)
- Change the order of rows (`arrange`)

Example data

```
head(mtcars, 3)
```

```
#>                mpg cyl disp  hp drat    wt  qsec vs am gear carb
#>  Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#>  Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#>  Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

- Car names are in an attribute of the data.frame called `rownames`. So it's not in a column
- We might want to convert fuel economy to metric
- We might just want to look at the relationship between displacement and fuel economy based on number of cylinders

---

The function `tibble::rownames_to_column` makes short work of the first point.

Row names are an older construct to give labels to each row. They are strongly discouraged outside of indices since using them as data created problems.

Example data (link)

```
link <- 'https://dl.dropboxusercontent.com/s/pqavhcckshqxtjm/brca.csv'
brca_data <- rio::import(link)
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
1	842302	M	17.990	10.38	122.80	1001.0	0.11840	0.27760	0.300100
2	842517	M	20.570	17.77	132.90	1326.0	0.08474	0.07864	0.086900
3	84300903	M	19.690	21.25	130.00	1203.0	0.10960	0.15990	0.197400
4	84348301	M	11.420	20.38	77.58	386.1	0.14250	0.28390	0.241400
5	84358402	M	20.290	14.34	135.10	1297.0	0.10030	0.13280	0.198000
6	843786	M	12.450	15.70	82.57	477.1	0.12780	0.17000	0.157800
7	844359	M	18.250	19.98	119.60	1040.0	0.09463	0.10900	0.112700
8	84458202	M	13.710	20.83	90.20	577.9	0.11890	0.16450	0.093660
9	844981	M	13.000	21.82	87.50	519.8	0.12730	0.19320	0.185900
10	84501001	M	12.460	24.04	83.97	475.9	0.11860	0.23960	0.227300
11	845636	M	16.020	23.24	102.70	797.8	0.08206	0.06669	0.032990
12	84610002	M	15.780	17.89	103.60	781.0	0.09710	0.12920	0.099540
13	846226	M	19.170	24.80	132.40	1123.0	0.09740	0.24580	0.206500
14	846381	M	15.850	23.95	103.70	782.7	0.08401	0.10020	0.099380
15	84667401	M	13.730	22.61	93.60	578.3	0.11310	0.22930	0.212800
16	84799002	M	14.540	27.54	96.73	658.8	0.11390	0.15950	0.163900
17	848406	M	14.680	20.13	94.74	684.5	0.09867	0.07200	0.073950
18	84862001	M	16.130	20.68	108.10	798.8	0.11700	0.20220	0.172200
19	849014	M	19.810	22.15	130.00	1260.0	0.09831	0.10270	0.147900

Showing 1 to 20 of 569 entries, 33 total columns

The `tidyverse` package

The `tidyverse` package is a meta-package that installs a set of packages that are useful for data cleaning, data tidying and data munging (manipulating data to get a computationally "attractive" dataset)

The `tidyverse` package

```
# install.packages('tidyverse')
library(tidyverse)
```

Core `tidyverse` packages

Package	Description
ggplot2	Data visualization
tibble	data.frame on steroids
tidyr	Data tidying (today)
readr	Reading text files (CSV)
purrr	Applying functions to data iteratively
dplyr	Data cleaning and munging (today)
stringr	String (character) manipulation
forcats	Manipulating categorical variables

Additional `tidyverse` packages

Package	Description
readxl	Read Excel files
haven	Read SAS, SPSS, Stata files
lubridate	Deal with dates and times
magrittr	Provides the pipe operator %>%
glue	Makes pasting text and data easier

Additional useful packages

Package	Description
broom	Turns the results of models or analysis into tidy datasets
fs	Allows directory and file manipulation in OS-agnostic manner
here	Allows robust specification of directory structure in a Project

Pipes

Pipes

Pipes (denoted %>%, spoken as "then") are to analytic pipelines as + is to `ggplot` layers

```
mpg1 <- mpg %>% mutate(id=1:n()) %>% select(id, year, trans, cty, hwy)
mpg_metric <- mpg1 %>%
  mutate(across(c(cty, hwy), function(x) {x * 1.6/3.8}))
```

Original data

id	year	trans	cty	hwy
1	1999	auto(l5)	18	29
2	1999	manual(m5)	21	29
3	2008	manual(m6)	20	31
4	2008	auto(av)	21	30
5	1999	auto(l5)	16	26

Transformed data

id	year	trans	cty	hwy
1	1999	auto(l5)	7.578947	12.21053
2	1999	manual(m5)	8.842105	12.21053
3	2008	manual(m6)	8.421053	13.05263
4	2008	auto(av)	8.842105	12.63158
5	1999	auto(l5)	6.736842	10.94737

Note I'm assigning a name to the transformed data. Otherwise it'll be lost

Verbs to use in pipes

The verbs in `tidyverse` are specially useful in pipes

Verb	Functionality
mutate	Transform a column with some function
select	Select some columns in the data
arrange	Order the data frame by values of a column(s)
filter	Keep only rows that meet some data criterion
group_by	Group by levels of a variable
pivot_longer	Transform a wide dataset to a long dataset
pivot_wider	Transform a long dataset to a wide dataset
separate	Separate one column into several columns
unite	Concatenate several columns into 1 column

Pipes almost always start with a data.frame/tibble object, and then "pipes" that data through different transformations (functions)

At each %>%, the results of the previous step are used as input for the next step.

A new verb: across

dplyr version 1.0 introduced a new verb, `across`. It supercedes the older scoped verbs `*_at`, `*_if` and `*_all`, where `*` might be `mutate` or `summarise`. Use whichever makes sense to you.

```
beaches %>%
  summarise_if(is.numeric, mean)
```

```
#>        year    month      day  season rainfall tem
#>  1 2015.494 6.377907 15.88953 2.52907       NA
#>    month_num
#>  1  36.30814
```

```
beaches %>%
  summarise(across(where(is.numeric), mean))
```

```
#>        year    month      day  season rainfall tem
#>  1 2015.494 6.377907 15.88953 2.52907       NA
#>    month_num
#>  1  36.30814
```

```
beaches %>%
  group_by(season_name) %>%
  summarise_at(vars(rainfall, enterococci),
               median)
```

```
#>  # A tibble: 4 x 3
#>    season_name rainfall enterococci
#>  * <chr>          <dbl>       <dbl>
#>  1 Autumn             0          NA
#>  2 Spring            NA          NA
#>  3 Summer             0          NA
#>  4 Winter             0          NA
```

```
beaches %>%
  group_by(season_name) %>%
  summarise(across(c(rainfall, enterococci),
                   median))
```

```
#>  # A tibble: 4 x 3
#>    season_name rainfall enterococci
#>  * <chr>          <dbl>       <dbl>
#>  1 Autumn             0          NA
#>  2 Spring            NA          NA
#>  3 Summer             0          NA
#>  4 Winter             0          NA
```

across (Updated Spring 2021)

The `across` function is now the more flexible standard way to apply most dplyr verbs across multiple columns, which can be selected the same way you would in `select`. In addition to what's on the previous slide, here are some more examples.

```
summaries <- list(
  mean = ~mean(.x, na.rm=T),
  median = ~median(.x, na.rm=T)
) # both the ~ and .x are required

beaches %>%
  summarise(across(.cols = where(is.numeric),
                   .fns = summaries))
```

```
#>    year_mean year_median month_mean month_median d
#>  1  2015.494        2016   6.377907            6 1
#>    rainfall_mean rainfall_median temperature_mean
#>  1      4.191228               0        23.57994
#>    enterococci_median day_num_mean day_num_median
#>  1           9.372727     178.7994            177
```

```
beaches %>%
  summarise(
    across(where(is.numeric), ~mean(.x, na.rm=T), .na
    across(where(is.numeric), ~median(.x, na.rm=T), .
```

```
#>    mean_year mean_month mean_day mean_season mean_
#>  1  2015.494   6.377907 15.88953     2.52907
#>    mean_day_num mean_month_num year month day seas
#>  1     178.7994       36.30814 2016     6  16
```

This groups the columns by function rather than by variable

across (Updated Spring 2021)

```
beaches %>%
  filter(across(everything(), ~!is.na(.x))) %>%
  head() # rows with no missing values
```

```
#>          date year month day season rainfall tempe
#>  1 2013-01-02 2013     1   2      1      0.0
#>  2 2013-01-06 2013     1   6      1      0.0
#>  3 2013-01-12 2013     1  12      1      0.0
#>  4 2013-01-18 2013     1  18      1      0.0
#>  5 2013-01-24 2013     1  24      1      0.0
#>  6 2013-01-30 2013     1  30      1      0.6
#>    season_name
#>  1      Summer
#>  2      Summer
#>  3      Summer
#>  4      Summer
#>  5      Summer
#>  6      Summer
```

For more examples see
https://dplyr.tidyverse.org/articles/colwise.html

Modeling and the broom package

Representing model relationships

In R, there is a particularly convenient way to express models, where you have

- one dependent variable
- one or more independent variables, with possible transformations and interactions

```
y ~ x1 + x2 + x1:x2 + I(x3^2) + x4*x5
```

Representing model relationships

```
y ~ x1 + x2 + x1:x2 + I(x3^2) + x4*x5
```

`y` depends on ...

- `x1` and `x2` linearly
- the interaction of `x1` and `x2` (represented as `x1:x2`)
- the square of `x3` (the `I()` notation ensures that the `^` symbol is interpreted correctly)
- `x4`, `x5` and their interaction (same as `x4 + x5 + x4:x5`)

Representing model relationships

```
y ~ x1 + x2 + x1:x2 + I(x3^2) + x4*x5
```

This interpretation holds for the vast majority of statistical models in R

- For decision trees and random forests and neural networks, don't add interactions or transformations, since the model will try to figure those out on their own

Our first model

```
library(survival)
myLinearModel <- lm(chol ~ bili, data = pbc)
```

Note that everything in R is an **object**, so you can store a model in a variable name.

This statement runs the model and stored the fitted model in `myLinearModel`

R does not interpret the model, evaluate the adequacy or appropriateness of the model, or comment on whether looking at the relationship between cholesterol and bilirubin makes any kind of sense.

Our first model

```
myLinearModel
```

```
#>
#>  Call:
#>  lm(formula = chol ~ bili, data = pbc)
#>
#>  Coefficients:
#>  (Intercept)         bili
#>       303.20        20.24
```

> Not very informative, is it?

Our first model

```
summary(myLinearModel)
```

```
#>
#>  Call:
#>  lm(formula = chol ~ bili, data = pbc)
#>
#>  Residuals:
#>      Min       1Q   Median       3Q      Max
#>  -565.39  -89.90  -35.36   44.92 1285.33
#>
#>  Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#>  (Intercept)  303.204     15.601  19.435  < 2e-16 ***
#>  bili          20.240      2.785   7.267 3.63e-12 ***
#>  ---
#>  Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#>  Residual standard error: 213.2 on 282 degrees of freedom
#>    (134 observations deleted due to missingness)
#>  Multiple R-squared:  0.1577,    Adjusted R-squared:  0.1547
#>  F-statistic:  52.8 on 1 and 282 DF,  p-value: 3.628e-12
```

> A little better

Our first model

```
broom::tidy(myLinearModel)
```

```
#>  # A tibble: 2 x 5
#>    term        estimate std.error statistic  p.value
#>    <chr>          <dbl>     <dbl>     <dbl>    <dbl>
#>  1 (Intercept)    303.      15.6      19.4  5.65e-54
#>  2 bili            20.2      2.79      7.27 3.63e-12
```

```
broom::glance(myLinearModel)
```

```
#>  # A tibble: 1 x 12
#>    r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
#>        <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
#>  1     0.158         0.155  213.      52.8 3.63e-12     1 -1925. 3856. 3867.
#>  # … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Displaying model results

Let's start with this model:

```
myModel <- lm(log10(enterococci) ~ rainfall + temperature + season_name + factor(year), data = beaches)
broom::tidy(myModel)
```

```
#>  # A tibble: 11 x 5
#>     term              estimate std.error statistic  p.value
#>     <chr>                <dbl>     <dbl>     <dbl>    <dbl>
#>   1 (Intercept)        1.23      0.208       5.92  8.39e- 9
#>   2 rainfall           0.0284    0.00296     9.58  2.72e-19
#>   3 temperature       -0.00830   0.00769    -1.08  2.81e- 1
#>   4 season_nameSpring -0.325     0.0840     -3.87  1.31e- 4
#>   5 season_nameSummer  0.0862    0.0909      0.948 3.44e- 1
#>   6 season_nameWinter -0.332     0.0889     -3.74  2.22e- 4
#>   7 factor(year)2014   0.0498    0.102       0.486 6.27e- 1
#>   8 factor(year)2015   0.0807    0.100       0.804 4.22e- 1
#>   9 factor(year)2016   0.0815    0.0975      0.836 4.04e- 1
#>  10 factor(year)2017  -0.0676    0.0972     -0.696 4.87e- 1
#>  11 factor(year)2018   0.0440    0.107       0.411 6.81e- 1
```

Displaying model results

Let's start with this model:

```
plt_data <- broom::tidy(myModel)
plt_data <- plt_data %>%
  filter(term != '(Intercept)') %>%
  mutate(term = str_replace(term,
                            'season_name',''))
plt_data
```

```
#>  # A tibble: 10 x 5
#>     term              estimate std.error statistic  p.value
#>     <chr>                <dbl>     <dbl>     <dbl>    <dbl>
#>   1 rainfall           0.0284    0.00296     9.58  2.72e-19
#>   2 temperature       -0.00830   0.00769    -1.08  2.81e- 1
#>   3 Spring            -0.325     0.0840     -3.87  1.31e- 4
#>   4 Summer             0.0862    0.0909      0.948 3.44e- 1
#>   5 Winter            -0.332     0.0889     -3.74  2.22e- 4
#>   6 factor(year)2014   0.0498    0.102       0.486 6.27e- 1
#>   7 factor(year)2015   0.0807    0.100       0.804 4.22e- 1
#>   8 factor(year)2016   0.0815    0.0975      0.836 4.04e- 1
#>   9 factor(year)2017  -0.0676    0.0972     -0.696 4.87e- 1
#>  10 factor(year)2018   0.0440    0.107       0.411 6.81e- 1
```

Displaying model results

Let's start with this model:

```
plt_data <- broom::tidy(myModel)
plt_data <- plt_data %>%
  filter(term != '(Intercept)') %>%
  mutate(term = str_replace(term,
                            'season_name','')) %>%
  mutate(term = str_replace(term,
                            'factor\\(year\\)','')) # Brackets are "escaped" using \\
plt_data
```

```
#>  # A tibble: 10 x 5
#>    term        estimate std.error statistic  p.value
#>    <chr>          <dbl>     <dbl>     <dbl>    <dbl>
#>  1 rainfall     0.0284    0.00296     9.58  2.72e-19
#>  2 temperature -0.00830   0.00769    -1.08  2.81e- 1
#>  3 Spring      -0.325     0.0840     -3.87  1.31e- 4
#>  4 Summer       0.0862    0.0909      0.948 3.44e- 1
#>  5 Winter      -0.332     0.0889     -3.74  2.22e- 4
#>  6 2014         0.0498    0.102       0.486 6.27e- 1
#>  7 2015         0.0807    0.100       0.804 4.22e- 1
#>  8 2016         0.0815    0.0975      0.836 4.04e- 1
#>  9 2017        -0.0676    0.0972     -0.696 4.87e- 1
#> 10 2018         0.0440    0.107       0.411 6.81e- 1
```

Displaying model results

Let's start with this model:

```
plt_data <- broom::tidy(myModel)
plt_data %>%
  filter(term != '(Intercept)') %>%
  mutate(term = str_replace(term,
                    'season_name','')) %>%
  mutate(term = str_replace(term,
                    'factor\\(year\\)','')) %>% # Bra
  ggplot(aes(x = term, y = estimate,
             ymin = estimate - 2 * std.error,
             ymax = estimate + 2 * std.error))+
    geom_pointrange()
```

Displaying model results

Let's start with this model:

```
plt_data <- broom::tidy(myModel)
plt_data %>%
  filter(term != '(Intercept)') %>%
  mutate(term = str_replace(term,
                  'season_name','')) %>%
  mutate(term = str_replace(term,
                  'factor\\(year\\)','')) %>% # Brack
  ggplot(aes(x = term, y = estimate,
             ymin = estimate - 2 * std.error,
             ymax = estimate + 2 * std.error))+
    geom_pointrange() +
    geom_hline(yintercept = 0, linetype=2) +
    theme_bw() +
    coord_flip()
```

Displaying model results

Let's start with this model:

```
plt_data <- broom::tidy(myModel)
plt_data %>%
  filter(term != '(Intercept)') %>%
  mutate(term = str_replace(term,
                  'season_name','')) %>%
  mutate(term = str_replace(term,
                  'factor\\(year\\)','')) %>% # Brack
  ggplot(aes(x = term, y = estimate,
             ymin = estimate - 2 * std.error,
             ymax = estimate + 2 * std.error))+
    geom_pointrange() +
    geom_hline(yintercept = 0, linetype=2) +
    theme_bw() +
    coord_flip()

ggsave('results.png') # ggsave knows format from file
```

You can also save the graph from the RStudio Plots pane, but coding it using `ggsave` is more reproducible

If you need to get a high-definition TIFF on a Mac, your best bet is to save your graph as a PDF and then convert it using Acrobat or other scripts (ask me if interested). The TIFF printer in R only creates 72 DPI TIFF files

Showing group differences ("Figure 1")

The package `ggpubr`, which extends `ggplot2`, makes this very easy. It provides the function `stat_compare_means`

Showing group differences ("Figure 1")

```
library(ggpubr)
theme_viz <- function(){
  theme_bw() +
    theme(axis.title = element_text(size=16),
          axis.text = element_text(size=14),
          text = element_text(size = 14))
}
ggplot(
  data=beaches,
  mapping= aes(x = season_name,
               y = log10(enterococci),
               color = season_name)) +
  geom_boxplot()+geom_jitter()+
  labs(x = 'Season',
       y = expression(paste('log'['10'],'(enterococci
       color='Season') +
  theme_viz()
```

Showing group differences ("Figure 1")

```
library(ggpubr)
plt <- ggplot(
  data=beaches,
  mapping= aes(x = season_name,
               y = log10(enterococci),
               color = season_name)) +
  geom_boxplot() +
  geom_jitter(width=0.1) +
  labs(x = 'Season',
       y = expression(paste('log'['10'],'(enterococci
       color='Season')+
  theme_viz()
my_comparisons <- list(c('Autumn','Spring'),
                       c('Spring','Summer'),
                       c('Summer','Winter'),
                       c('Spring','Winter'))
plt + stat_compare_means()
```

Showing group differences ("Figure 1")

```
library(ggpubr)
plt <- ggplot(
  data=beaches,
  mapping= aes(x = season_name,
               y = log10(enterococci),
               color = season_name)) +
  geom_boxplot() +
  geom_jitter(width=0.1)+
    labs(x = 'Season',
         y = expression(paste('log'['10'],'(enterococ
         color='Season') +
  theme_viz()
my_comparisons <- list(c('Autumn','Spring'),
                       c('Spring','Summer'),
                       c('Summer','Winter'),
                       c('Spring','Winter'))
plt + stat_compare_means() +
  stat_compare_means(comparisons = my_comparisons)
```

Showing group differences ("Figure 1")

```
library(ggpubr)
plt <- ggplot(
  data=beaches,
  mapping= aes(x = season_name,
               y = log10(enterococci),
               color = season_name)) +
  geom_boxplot() +
  geom_jitter(width=0.1)+
  labs(x = 'Season',
       y = expression(paste('log'['10'],'(enterococci
  theme_viz()
my_comparisons <- list(c('Autumn','Spring'),
                       c('Spring','Summer'),
                       c('Summer','Winter'),
                       c('Spring','Winter'))
plt + stat_compare_means(label.y = 6) +
  stat_compare_means(comparisons = my_comparisons)
```

Manipulating data for plotting

We would like to get density plots of all the variables

```
dat_spine <- rio::import('data/Dataset_spine.csv',
                         check.names=T)
head(dat_spine)
```

```
#>    pelvic_incidence pelvic_tilt lumbar_lordosis_ar
#>  1         63.02782   22.552586              39.60
#>  2         39.05695   10.060991              25.01
#>  3         68.83202   22.218482              50.09
#>  4         69.29701   24.652878              44.31
#>  5         49.71286    9.652075              28.31
#>  6         40.25020   13.921907              25.12
#>    degree_spondylolisthesis pelvic_slope direct_ti
#>  1                -0.254400    0.7445035     12.56
#>  2                 4.564259    0.4151857     12.88
#>  3                -3.530317    0.4748892     26.83
#>  4                11.211523    0.3693453     23.56
#>  5                 7.918501    0.5433605     35.49
#>  6                 2.230652    0.7899929     29.32
#>    cervical_tilt sacrum_angle scoliosis_slope clas
#>  1      15.30468   -28.658501         43.5123
#>  2      16.78486   -25.530607         16.1102
#>  3      16.65897   -29.031888         19.2221
#>  4      11.42447   -30.470246         18.8329
#>  5       8.87237   -16.378376         24.9171
```

Facets only work by grouping on a variable. Here we have data in several columns

Manipulating data for plotting

We would like to get density plots of all the variables.

```
dat_spine %>%
  tidyr::gather(variable, value, everything())
```

```
#>                      variable        value
#>  1           pelvic_incidence   63.0278175
#>  2           pelvic_incidence  39.05695098
#>  3           pelvic_incidence  68.83202098
#>  4           pelvic_incidence  69.29700807
#>  5           pelvic_incidence  49.71285934
#>  6           pelvic_incidence  40.25019968
#>  7           pelvic_incidence  53.43292815
#>  8           pelvic_incidence  45.36675362
#>  9           pelvic_incidence  43.79019026
#>  10          pelvic_incidence  36.68635286
#>  11          pelvic_incidence  49.70660953
#>  12          pelvic_incidence  31.23238734
#>  13          pelvic_incidence  48.91555137
#>  14          pelvic_incidence   53.5721702
#>  15          pelvic_incidence  57.30022656
#>  16          pelvic_incidence  44.31890674
#>  17          pelvic_incidence  63.83498162
#>  18          pelvic_incidence  31.27601184
#>  19          pelvic_incidence  38.69791243
#>  20          pelvic_incidence  41.72996308
```

The `gather` function turns this wide dataset to a long dataset, stacking all the variables on top of each other

Manipulating data for plotting

We would like to get density plots of all the variables.

```
dat_spine %>%
  select(pelvic_incidence:sacral_slope) %>%
  tidyr::gather(variable, value) %>%
  ggplot(aes(x = value)) +
  geom_density() +
  facet_wrap(~variable) +
  labs(x = '')
```

This is one of my most used tricks for getting facetted plots from wide data

Re-ordering factors

```
beaches %>%
  ggplot(aes(x = season_name, y = temperature)) +
  geom_boxplot() +
  scale_y_continuous(labels =
        scales::unit_format(unit = "\u00B0C")) +
  labs(x = 'Season', y = 'Temperature') +
  theme_bw() +
  theme(axis.title = element_text(size = 16),
        axis.text = element_text(size = 14))
```

Re-ordering factors

```
beaches %>%
  mutate(season_name =
      fct_relevel(season_name,
        'Autumn','Winter','Spring','Summer')) %>%
  ggplot(aes(x = season_name, y = temperature)) +
  geom_boxplot() +
  scale_y_continuous(labels =
        scales::unit_format(unit = "\u00B0C")) +
  labs(x = 'Season', y = 'Temperature') +
  theme_bw() +
  theme(axis.title = element_text(size = 16),
        axis.text = element_text(size = 14))
```