Data manipulation and visualisation with R

RS-eco

Biodiversity & Global Change Lab
Technical University of Munich, Germany

rs-eco@posteo.de

08th April 2019

Piping with magrittr

The common way of coding in R

```
mean_length <- mean(iris$Sepal.Length)
sub_iris <- subset(iris, Sepal.Length > mean_length)
cor_iris <- cor(sub_iris$Sepal.Length,
                sub_iris$Sepal.Width)
cor_iris

## [1] 0.3361992
```

The new way of coding in R

```
iris %>%
  subset(Sepal.Length > mean(Sepal.Length)) %$%
  cor(Sepal.Length, Sepal.Width)

## [1] 0.3361992
```

magrittr - Pipes

The forward-pipe operator, **%>%** can be used to replace f(x) with x %>% f or f(x,y) with x %>% f(y).

```
# Normal way
rnorm(5)

# Using piping

# With round brackets
5 %>% rnorm()

# Or without round brackets
5 %>% rnorm
```

magrittr - Pipes

The compound assignment pipe-operator, %<>% can be used whenever expr <- ... makes sense, e.g.

```
# Normal way
x <- rnorm(5)
x <- sort(x)

# Using piping
x <- rnorm(5)
x %<>% sort
```

magrittr - Pipes

The "tee" operator, **%T>%** works like **%>%**, except it returns the left-hand side value, and not the result of the right-hand side operation. This is useful when a step in a pipeline is used for its side-effect (printing, plotting, logging, etc.). As an example (where the actual plot is omitted here):

```
rnorm(200) %>%
matrix(ncol = 2) %T>%
plot %>% # plot usually does not return anything.
colSums
```

Note: Pipe operators can never stand at the beginning of a line.

magrittr - Pipes

The "exposition" pipe operator, **%$%** exposes the names within the left-hand side object to the right-hand side expression. This operator is handy when functions do not themselves have a data argument:

```
# Normal way
cor(iris$Sepal.Length, iris$Sepal.Width)

# Using piping
iris %$% cor(Sepal.Length, Sepal.Width)
```

Do it yourself - Exercise 1

- Install and load the **magrittr** package
- Turn the following code into a streamlined version using piping:

```
mean_width <- mean(iris$Sepal.Width)
iris_sub <- subset(iris, Sepal.Width < mean_width)
plot(iris_sub)
cor_iris_sub <- cor(iris_sub$Sepal.Length,
                    iris_sub$Sepal.Width)
```

Data import with readr

Read and write data

Reading and Writing Data

Also see the **readr** package.

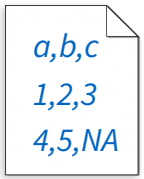
Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

readr - Read functions

Read tabular data to tibbles

These functions share the common arguments:

read_*(file, col_names = TRUE, col_types = NULL, locale = default_locale(), na = c("", "NA"), quoted_na = TRUE, comment = "", trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000, n_max), progress = interactive()**)**

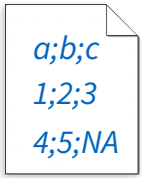


A	B	C
1	2	3
4	5	NA

read_csv()

Reads comma delimited files.

read_csv("file.csv")

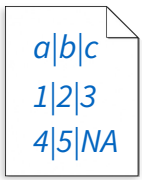


A	B	C
1	2	3
4	5	NA

read_csv2()

Reads Semi-colon delimited files.

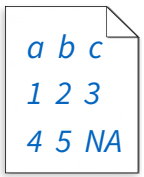
read_csv2("file2.csv")



A	B	C
1	2	3
4	5	NA

read_delim(delim, quote = "\"", escape_backslash = FALSE, escape_double = TRUE**)** Reads files with any delimiter.

read_delim("file.txt", delim = "|")



A	B	C
1	2	3
4	5	NA

read_fwf(col_positions**)**

Reads fixed width files.

read_fwf("file.fwf", col_positions = c(1, 3, 5))

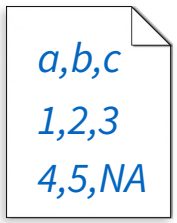
read_tsv()

Reads tab delimited files. Also **read_table().**

read_tsv("file.tsv")

readr - Read arguments

Useful arguments



Example file

write_csv (path = "file.csv",
x = read_csv("a,b,c\n1,2,3\n4,5,NA"))

A	B	C
1	2	3
4	5	NA

No header

read_csv("file.csv",
***col_names = FALSE**)*

x	y	z
A	B	C
1	2	3
4	5	NA

Provide header

read_csv("file.csv",
***col_names = c("x", "y", "z")**)*

1	2	3
4	5	NA

Skip lines

read_csv("file.csv",
***skip = 1**)*

A	B	C
1	2	3

Read in a subset

read_csv("file.csv",
***n_max = 1**)*

A	B	C
1	2	3
NA	NA	NA

Missing Values

read_csv("file.csv",
***na = c("4", "5", ".")**)*

readr - Write functions

Save **x**, an R object, to **path**, a filepath, with:

write_csv(x, path, na = "NA", append = FALSE, col_names = !append**)**

Tibble/df to comma delimited file.

write_delim(x, path, delim = " ", na = "NA", append = FALSE, col_names = !append**)**

Tibble/df to file with any delimiter.

write_excel_csv(x, path, na = "NA", append = FALSE, col_names = !append**)**

Tibble/df to a CSV for excel

write_file(x, path, append = FALSE**)**

String to file.

write_lines(x, path, na = "NA", append = FALSE**)**

String vector to file, one element per line.

write_rds(x, path, compress = c("none", "gz", "bz2", "xz"), ...**)**

Object to RDS file.

write_tsv(x, path, na = "NA", append = FALSE, col_names = !append**)**

Tibble/df to tab delimited files.

write_csv() can save compressed files, just add the compression type to the file ending (i.e. species_info.csv.xz). The compressed files can be directly read into R, using **read.csv()** and **read_csv()**.

Other types of data

The following packages can be used to import other types of files:

- **haven** - SPSS, Stata and SAS files
- **readxl** - excel files (.xls and .xlsx)
- **dplyr** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **raster, stars** - raster files (.nc, .ascii, .tif)
- **rgdal, sf** - vector files (.shp)

Do it yourself - Exercise 2

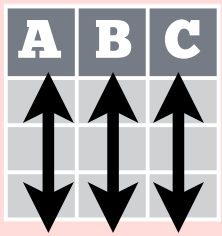
- Install and load the **readr** package
- Read the species_info.csv.xz file into R
- Read only the first 10 rows of your data into R
- Read the last 10 rows of your data into R
- Check the structure and class of your object

Data handling with tidyr

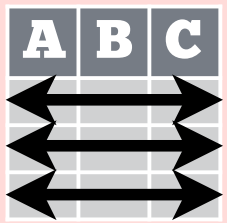
Tidy Data with tidyr

Tidy data is a way to organize tabular data. It provides a consistent data structure across packages.

A table is tidy if:



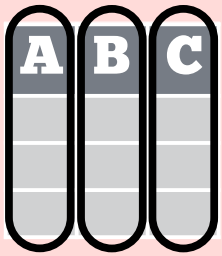
&



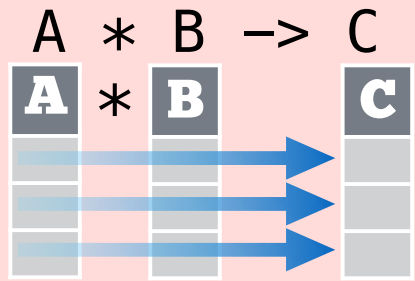
Each **variable** is in its own **column**

Each **observation**, or **case**, is in its own **row**

Tidy data:



Makes variables easy to access as vectors



Preserves cases during vectorized operations

Is species_info.csv.xz tidy?

Data layout

tidy data typically comes in one of two layouts:

A **wide table** or a **long table**

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout. Each uses the idea of a key column: value column pair.

gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE**)**

Gather moves column names into a key column, gathering the column values into a single value column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL**)**

Spread moves the unique values of a key column into the column names, spreading the values of a value column across the new columns that result.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

key value

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

spread(table2, type, count)

tidyr - Handle missing values

Handle Missing Values

drop_na(data, ...)

Drop rows containing NA's in … columns.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, ..., .direction = c("down", "up"))

Fill in NA's in … columns with most recent non-NA values.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), ...)

Replace NA's by column.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	2
C	2
D	3
E	2

replace_na(x,list(x2 = 2), x2)

tidyr - Expand tables

Expand Tables - quickly create tables with combinations of values

complete(data, ..., fill = list()**)**

Adds to the data missing combinations of the values of the variables listed in …

complete(mtcars, cyl, gear, carb)

expand(data, ...**)**

Create new tibble with all possible combinations of the values of the variables listed in …

expand(mtcars, cyl, gear, carb)

tidyr - Split and combine cells

Use these functions to split or combine cells into individual, isolated values

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...**)**

Separate each cell in a column to make several columns.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

separate_rows(table3, rate, into = c("cases", "pop"))

unite(data, col, ..., sep = "_", remove = TRUE**)**

Collapse cells across several columns to make a single column.

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

→

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

unite(table5, century, year, col = "year", sep = "")

Do it yourself - Exercise 3

Using the smallest amount of code:

- Install and load the **tidyr** package
- Replace NAs in presence, origin and season with 0s
- Complete species_info by binomial, presence, origin and season
- Drop missing values from species_info
- Separate binomial into genus and species, but keep binomial
- Read species_data.csv.xz into R
- Turn species_data from wide into long format and drop missing values
- Turn species_data back into wide format

Don't forget to use pipes

Data transformation with dplyr

Summarise Cases

These apply **summary functions** to columns to create a new table. Summary functions take vectors as input and return one value (see back).

summarise(.data, …**)**
Compute table of summaries. Also **summarise_()**.
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE**)**
Count number of rows in each group defined by the variables in … Also **tally()**.
count(iris, Species)

Variations

- **summarise_all()** - Apply funs to every column.
- **summarise_at()** - Apply funs to specific columns.
- **summarise_if()** - Apply funs to all cols of one type.

dplyr - Summarise functions

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

Counts

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

Location

mean() - mean, also **mean(!is.na())**
median() - median

Logicals

mean() - Proportion of TRUE's
sum() - # of TRUE's

Position/Order

dplyr::**first()** - first value
dplyr::**last()** - last value
dplyr::**nth()** - value in nth location of vector

Rank

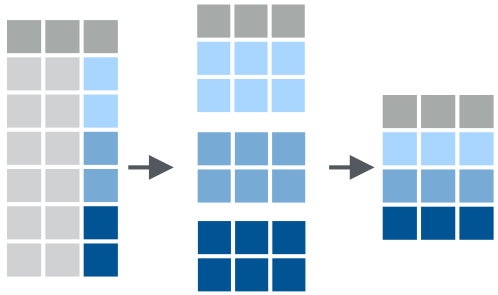
quantile() - nth quantile
min() - minimum value
max() - maximum value

Spread

IQR() - Inter-Quartile Range
mad() - mean absolute deviation
sd() - standard deviation
var() - variance

Group Cases

Use **group_by()** to created a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.



```
mtcars %>%
 group_by(cyl) %>%
  summarise(avg = mean(mpg))
```

group_by(.data, ..., add = FALSE**)**

Returns copy of table grouped by …

g_iris <- group_by(iris, Species)

ungroup(x, ...**)**

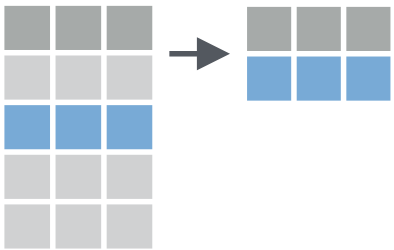
Returns ungrouped copy of table.

ungroup(g_iris)

Note: group_by() can be used with multiple variables, i.e. group_by(x,y).

dplyr - Extract cases

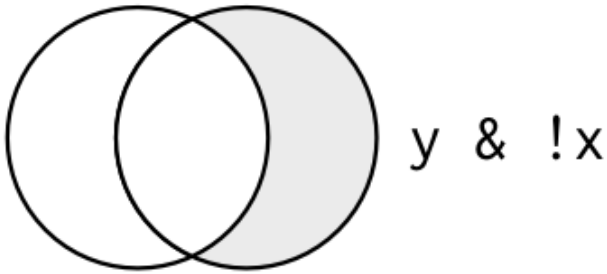
Extract certain rows of your data.frame using filter()



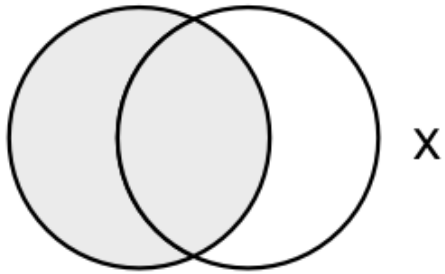
filter(.data, ...**)**
Extract rows that meet logical criteria. Also **filter_()**. *filter(iris, Sepal.Length > 7)*

You can use filter() with logical operators (>, <, >=, <=), is.na(), !is.na() or boolean operators.

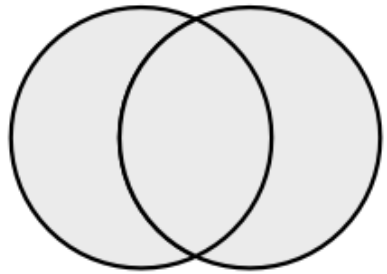
Boolean operators



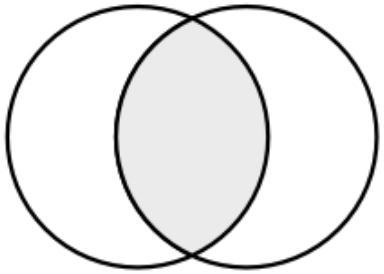
y & !x



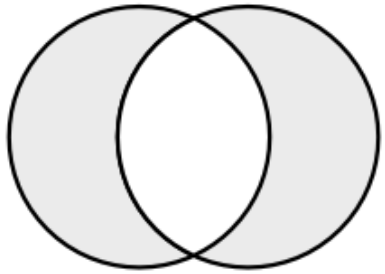
x



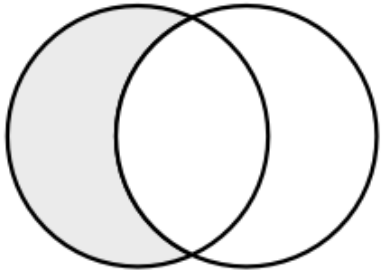
x | y



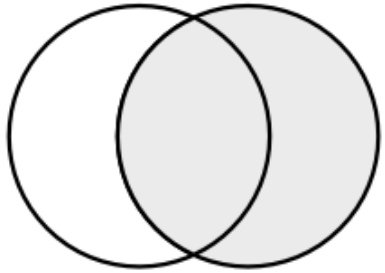
x & y



xor(x, y)



x & !y

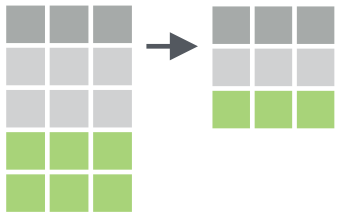


y

Also check out **%in%**, the best operator ever!!!

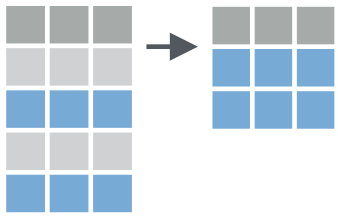
dplyr - Extract cases

Row functions return a subset of rows as a new table.



distinct(.data, ..., .keep_all = FALSE**)**

Remove rows with duplicate values. Also **distinct_()**. *distinct(iris, Species)*



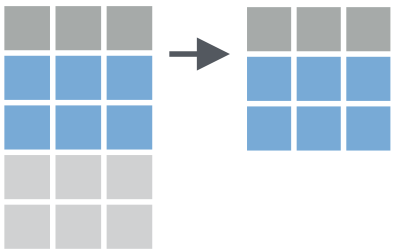
sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()**)**

Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()**)**

Randomly select size rows.
sample_n(iris, 10, replace = TRUE)

dplyr - Extract cases II



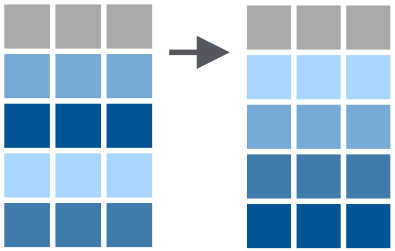
slice(.data, . . .**)**

Select rows by position. Also **slice_()**. *slice(iris, 10:15)*

top_n(*x*, *n*, *wt***)**

Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Arrange Cases

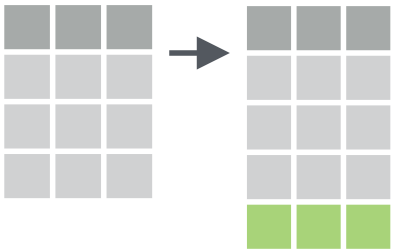


arrange(.data, ...**)**

Order rows by values of a column (low to high), use with **desc()** to order from high to low.

arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

Add Cases



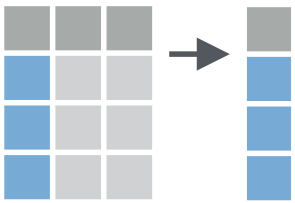
add_row(.data, ..., .before = NULL, .after = NULL**)**

Add one or more rows to a table.

add_row(faithful, eruptions = 1, waiting = 1)

dplyr - Extract Variables

Column functions return a set of columns as a new table.



select(.data, . . .**)**

Extract columns by name. Also **select_if()**

select(iris, Sepal.Length, Species)

Use these helpers with select(),

e.g. *select(iris, starts_with("Sepal"))*

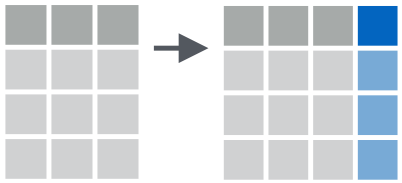
contains(match**)**
ends_with(match**)**
matches(match**)**

num_range(prefix, range**)**
one_of(. . .**)**
starts_with(match**)**

:, e.g. mpg:cyl
-, e.g, -Species

dplyr - Make New Variables

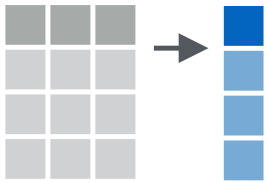
These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).



mutate(.data, …**)**

Compute new column(s).

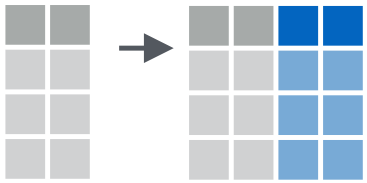
mutate(mtcars, gpm = 1/mpg)



transmute(.data, …**)**

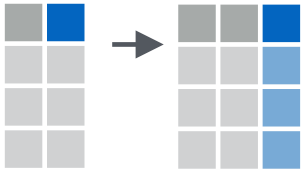
Compute new column(s), drop others.

transmute(mtcars, gpm = 1/mpg)



mutate_all(.tbl, .funs, ...**)**

Apply funs to every column. Use with **funs()**. *mutate_all(faithful, funs(log(.), log2(.)))*



mutate_at(.tbl, .cols, .funs, ...**)**

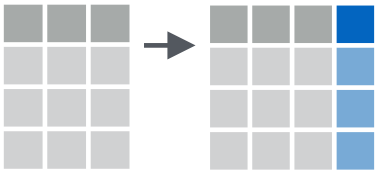
Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().

mutate_at(iris, vars(-Species), funs(log(.)))

mutate_if(.tbl, .predicate, .funs, ...**)**

Apply funs to all columns of one type. Use with **funs().**

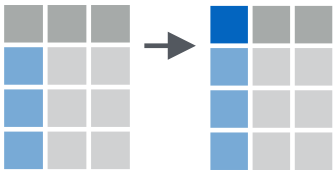
mutate_if(iris, is.numeric, funs(log(.)))



add_column(.data, ..., .before = NULL, .after = NULL**)**

Add new column(s).

add_column(mtcars, new = 1:32)



rename(.data, ...**)**

Rename columns.

rename(iris, Length = Sepal.Length)

dplyr - Vectorized functions

mutate() and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

Offsets

dplyr::**lag()** - Offset elements by 1

dplyr::**lead()** - Offset elements by -1

Cumulative Aggregates

dplyr::**cumall()** - Cumulative all()

dplyr::**cumany()** - Cumulative any()

cummax() - Cumulative max()

dplyr::**cummean()** - Cumulative mean()

cummin() - Cumulative min()

cumprod() - Cumulative prod()

cumsum() - Cumulative sum()

Rankings

dplyr::**cume_dist()** - Proportion of all values <=

dplyr::**dense_rank()** - rank with ties = min, no gaps

dplyr::**min_rank()** - rank with ties = min

dplyr::**ntile()** - bins into n bins

dplyr::**percent_rank()** - min_rank scaled to [0,1]

dplyr::**row_number()** - rank with ties = "first"

Math

+, **-** , *****, **/**, **^**, **%/%**, **%%** - arithmetic ops

log(), **log2()**, **log10()** - logs

<, **<=**, **>**, **>=**, **!=**, **==** - logical comparisons

Do it yourself - Exercise 4

Using pipes:

- Calculate the number of occurrences per species
- Arrange data by the number of occurrences per species
- Identify all species with less than 10 occurrences
- Calculate the number of species per grid cell
- Extract data for five species using %in%
- Extract data for all species starting with "Ac"
- Calculate the number of species per class in species_info and output as table using the **kable()** function of the **knitr** package

Data transformation with dplyr - Part II

dplyr - Combine variables

x

A	B	C
a	t	1
b	u	2
c	v	3

+

y

A	B	D
a	t	3
b	u	2
d	w	1

=

Use **bind_cols()** to paste tables beside each other as they are.

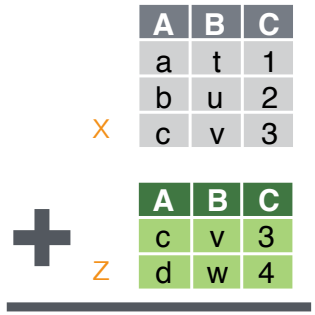
A	B	C	A	B	D
a	t	1	a	t	3
b	u	2	b	u	2
c	v	3	d	w	1

bind_cols(...)

Returns tables placed side by side as a single table.

bind_cols() also works with many (>2) data frames, and is an efficient implementation of **do.call(cbind, dfs)**.

dplyr - Combine cases



A	B	C
a	t	1
b	u	2
c	v	3

A	B	C
c	v	3
d	w	4

Use **bind_rows()** to paste tables below each other as they are.

DF	A	B	C
x	a	t	1
x	b	u	2
x	c	v	3
z	c	v	3
z	d	w	4

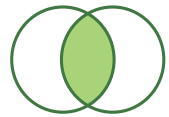
bind_rows(..., .id = NULL**)**

Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

A	B	C
c	v	3

intersect(x, y, ...**)**

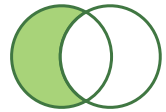
Rows that appear in both x and z.



A	B	C
a	t	1
b	u	2

setdiff(x, y, ...**)**

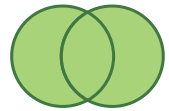
Rows that appear in x but not z.



A	B	C
a	t	1
b	u	2
c	v	3
d	w	4

union(x, y, ...**)**

Rows that appear in x or z. (Duplicates removed). **union_all()** retains duplicates.



Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

bind_rows() also works with many (>2) data frames, and is an efficient implementation of **do.call(rbind, dfs)**.

dplyr - Mutating Join

Use a "Mutating Join" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...**)**

Join matching values from y to x.

A	B	C	D
a	t	1	3
b	u	2	2
d	w	NA	1

right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...**)**

Join matching values from x to y.

dplyr - Mutating Join II

Use a "Mutating Join" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

A	B	C	D
a	t	1	3
b	u	2	2

inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...**)**

Join data. Retain only rows with matches.

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA
d	w	NA	1

full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...**)**

Join data. Retain all values, all rows.

dplyr - Mutating Join III

A	B.x	C	B.y	D
a	t	1	t	3
b	u	2	u	2
c	v	3	NA	NA

Use **by = c("col1", "col2")** to specify the column(s) to match on.

left_join(x, y, by = "A")

A.x	B.x	C	A.y	B.y
a	t	1	d	w
b	u	2	b	u
c	v	3	a	t

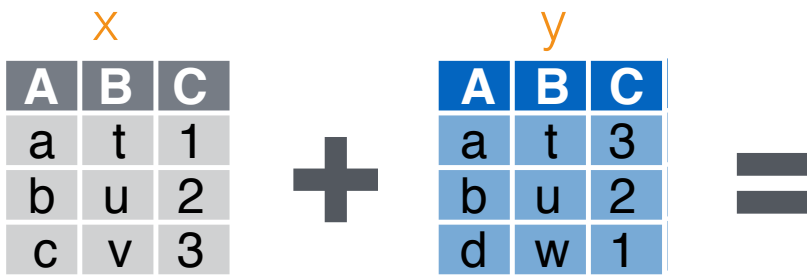
Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.

left_join(x, y, by = c("C" = "D"))

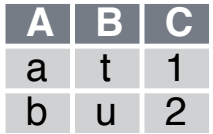
But what, if you want to join a list of data frames?

```
# Combine dplyr join commands, with Reduce
Reduce(function(...) dplyr::left_join(..., all.x=TRUE), dfs)
```

dplyr - Extract rows



Use a "**Filtering Join**" to filter one table against the rows of another.



semi_join(x, y, by = NULL, . . .**)**

Return rows of x that have a match in y. USEFUL TO SEE WHAT WILL BE JOINED.

anti_join(x, y, by = NULL, . . .**)**

Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

Do it yourself - Exercise 5

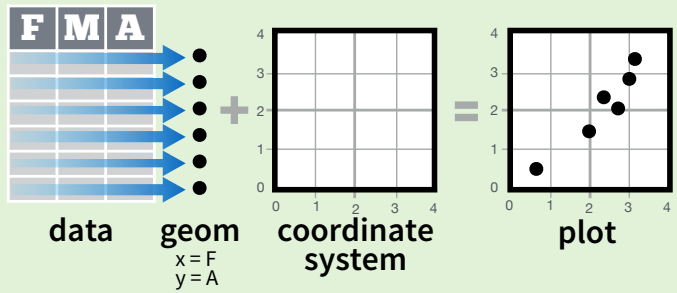
Using pipes:

- Join species_data with species_info
- Calculate the number of species per class in species_data
- Calculate the number of species per grid cell for each class
- Identify the most widely distributed species for each class

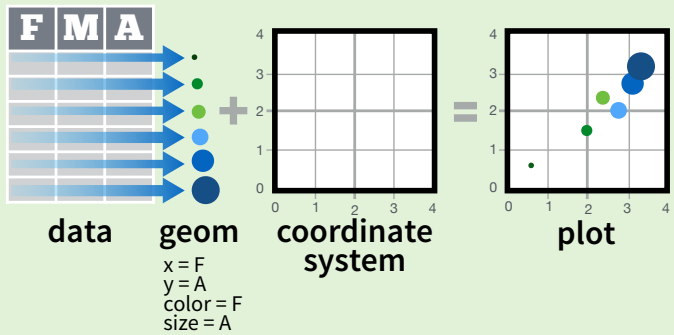
Data visualization with ggplot2

ggplot2 - Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same few components: a **data** set, a set of **geoms**—visual marks that represent data points, and a **coordinate system.**



To display data values, map variables in the data set to aesthetic properties of the geom like **size**, **color**, and **x** and **y** locations.



ggplot2 - Basics II

Build a graph with **ggplot()** or **qplot()**

ggplot(data = mpg, **aes(**x = cty, y = hwy**))**

Begins a plot that you finish by adding layers to. No defaults, but provides more control than qplot().

data

```
ggplot(mpg, aes(hwy, cty)) +
 geom_point(aes(color = cyl)) +
 geom_smooth(method ="lm") +
 coord_cartesian() +
 scale_color_gradient() +
 theme_bw()
```

add layers, elements with +

layer = geom + default stat + layer specific mappings

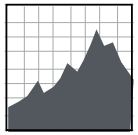
additional elements

Add a new layer to a plot with a **geom_*()** or **stat_*()** function. Each provides a geom, a set of aesthetic mappings, and a default stat and position adjustment.

ggplot2 - One variable

Continuous

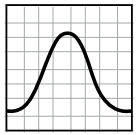
a <- ggplot(mpg, aes(hwy))



a + geom_area(stat = "bin")

x, y, alpha, color, fill, linetype, size

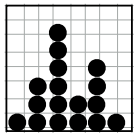
b + geom_area(aes(y = ..density..), stat = "bin")



a + geom_density(kernel = "gaussian"**)**

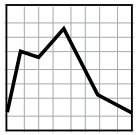
x, y, alpha, color, fill, linetype, size, weight

b + geom_density(aes(y = ..county..))



a + geom_dotplot()

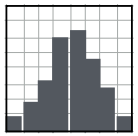
x, y, alpha, color, fill



a + geom_freqpoly()

x, y, alpha, color, linetype, size

b + geom_freqpoly(aes(y = ..density..))



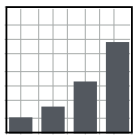
a + geom_histogram(binwidth = 5**)**

x, y, alpha, color, fill, linetype, size, weight

b + geom_histogram(aes(y = ..density..))

Discrete

b <- ggplot(mpg, aes(fl))



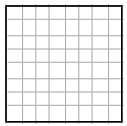
b + geom_bar()

x, alpha, color, fill, linetype, size, weight

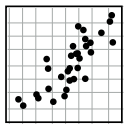
ggplot2 - Two variables

Continuous X, Continuous Y

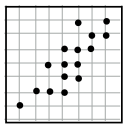
f <- ggplot(mpg, aes(cty, hwy))



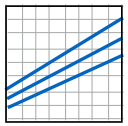
f + geom_blank()
(Useful for expanding limits)



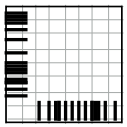
f + geom_jitter()
x, y, alpha, color, fill, shape, size



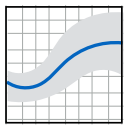
f + geom_point()
x, y, alpha, color, fill, shape, size



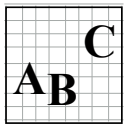
f + geom_quantile()
x, y, alpha, color, linetype, size, weight



f + geom_rug(sides = "bl"**)**
alpha, color, linetype, size



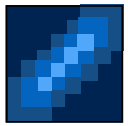
f + geom_smooth(method = lm**)**
x, y, alpha, color, fill, linetype, size, weight



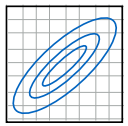
f + geom_text(aes(label = cty)**)**
x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

Continuous Bivariate Distribution

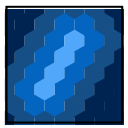
i <- ggplot(movies, aes(year, rating))



i + geom_bin2d(binwidth = c(5, 0.5)**)**
xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size, weight



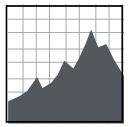
i + geom_density2d()
x, y, alpha, colour, linetype, size



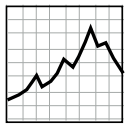
i + geom_hex()
x, y, alpha, colour, fill size

Continuous Function

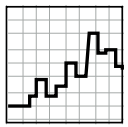
j <- ggplot(economics, aes(date, unemploy))



j + geom_area()
x, y, alpha, color, fill, linetype, size



j + geom_line()
x, y, alpha, color, linetype, size

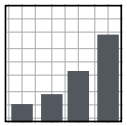


j + geom_step(direction = "hv"**)**
x, y, alpha, color, linetype, size

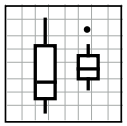
ggplot2 - Two variables II

Discrete X, Continuous Y

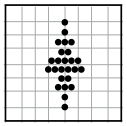
g <- ggplot(mpg, aes(class, hwy))



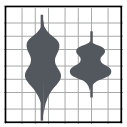
g + geom_bar(stat = "identity")
x, y, alpha, color, fill, linetype, size, weight



g + geom_boxplot()
lower, middle, upper, x, ymax, ymin, alpha, color, fill, linetype, shape, size, weight



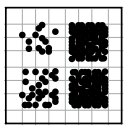
g + geom_dotplot(binaxis = "y", stackdir = "center"**)**
x, y, alpha, color, fill



g + geom_violin(scale = "area"**)**
x, y, alpha, color, fill, linetype, size, weight

Discrete X, Discrete Y

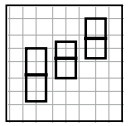
h <- ggplot(diamonds, aes(cut, color))



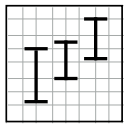
h + geom_jitter()
x, y, alpha, color, fill, shape, size

Visualizing error

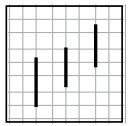
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
k <- ggplot(df, aes(grp, fit, ymin = fit-se, ymax = fit+se))



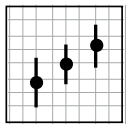
k + geom_crossbar(fatten = 2**)**
x, y, ymax, ymin, alpha, color, fill, linetype, size



k + geom_errorbar()
x, ymax, ymin, alpha, color, linetype, size, width (also **geom_errorbarh()**)



k + geom_linerange()
x, ymin, ymax, alpha, color, linetype, size



k + geom_pointrange()
x, y, ymin, ymax, alpha, color, fill, linetype, shape, size

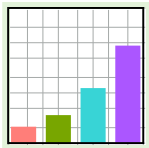
Maps

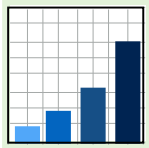
data <- data.frame(murder = USArrests$Murder,
state = tolower(rownames(USArrests)))
map <- map_data("state")
l <- ggplot(data, aes(fill = murder))

l + geom_map(aes(map_id = state), map = map**) + expand_limits(**x = map$long, y = map$lat**)**
map_id, alpha, color, fill, linetype, size

ggplot2 - Scales

Scales control how a plot maps data values to the visual values of an aesthetic. To change the mapping, add a custom scale.





```
n <- b + geom_bar(aes(fill = fl))
n
```

scale_ | aesthetic to adjust | prepackaged scale to use | scale specific arguments

```
n + scale_fill_manual(
  values = c("skyblue", "royalblue", "blue", "navy"),
  limits = c("d", "e", "p", "r"), breaks =c("d", "e", "p", "r"),
  name = "fuel", labels = c("D", "E", "P", "R"))
```

range of values to include in mapping | title to use in legend/axis | labels to use in legend/axis | breaks to use in legend/axis

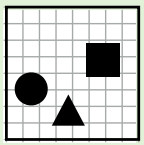
General Purpose scales

Use with any aesthetic:
alpha, color, fill, linetype, shape, size

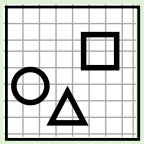
scale_*_continuous() - map cont' values to visual values
scale_*_discrete() - map discrete values to visual values
scale_*_identity() - use data values **as** visual values
scale_*_manual(values = c()**)** - map discrete values to manually chosen visual values

ggplot2 - Scales II

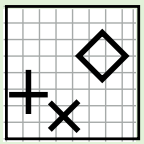
Shape scales



p <- f + geom_point(
aes(shape = fl)**)**

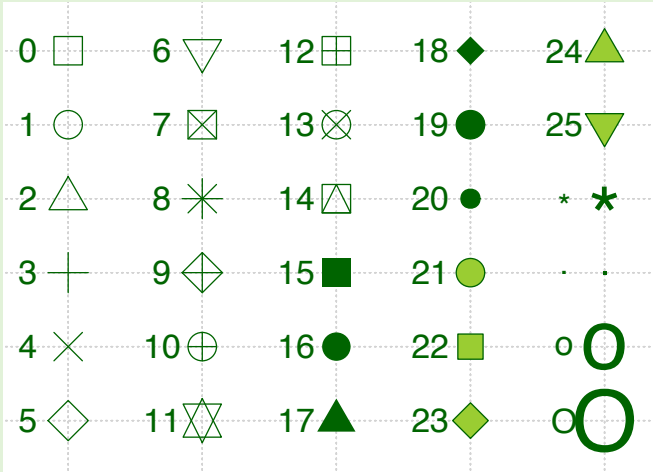


p + scale_shape(
solid = FALSE**)**

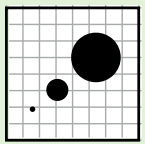


p + scale_shape_manual(
values = c(3:7)**)**
Shape values shown in chart on right

Manual shape values



Size scales

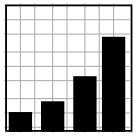


q <- f + geom_point(
aes(size = cyl)**)**

q + scale_size_area(max = 6**)**
Value mapped to area of circle (not radius)

ggplot2 - Coordinate systems

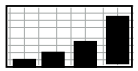
```
r <- b + geom_bar()
```



r + coord_cartesian(xlim = c(0, 5)**)**

xlim, ylim

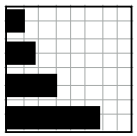
The default cartesian coordinate system



r + coord_fixed(ratio = 1/2**)**

ratio, xlim, ylim

Cartesian coordinates with fixed aspect ratio between x and y units



r + coord_flip()

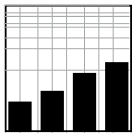
xlim, ylim

Flipped Cartesian coordinates

r + coord_polar(theta = "x", direction=1 **)**

theta, start, direction

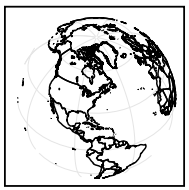
Polar coordinates



r + coord_trans(ytrans = "sqrt"**)**

xtrans, ytrans, limx, limy

Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.



z + coord_map(projection = "ortho", orientation=c(41, -74, 0)**)**

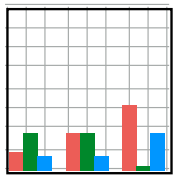
projection, orientation, xlim, ylim

Map projections from the mapproj package (mercator (default), azequalarea, lagrange, etc.)

Position adjustments

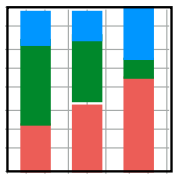
Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

```
s <- ggplot(mpg, aes(fl, fill = drv))
```



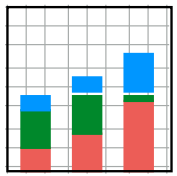
s + geom_bar(position = "dodge")

Arrange elements side by side



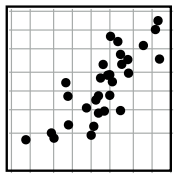
s + geom_bar(position = "fill")

Stack elements on top of one another, normalize height



s + geom_bar(position = "stack")

Stack elements on top of one another



f + geom_point(position = "jitter")

Add random noise to X and Y position of each element to avoid overplotting

Each position adjustment can be recast as a function with manual **width** and **height** arguments

Labels & Legends

Labels

t + ggtitle("New Plot Title")
Add a main title above the plot

t + xlab("New X label")
Change the label on the X axis

t + ylab("New Y label")
Change the label on the Y axis

t + labs(title =" New title", x = "New x", y = "New y")
All of the above

Use scale functions to update legend labels

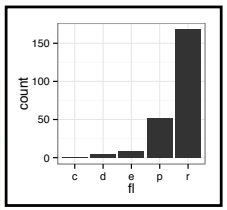
Legends

t + theme(legend.position = "bottom")
Place legend at "bottom", "top", "left", or "right"

t + guides(color = "none")
Set legend type for each aesthetic: colorbar, legend, or none (no legend)

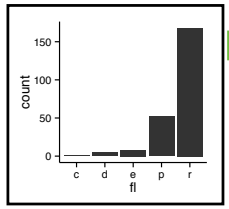
t + scale_fill_discrete(name = "Title", labels = c("A", "B", "C"))
Set legend title and labels with a scale function.

Themes



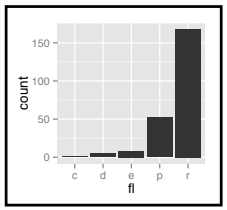
r + theme_bw()

White background with grid lines



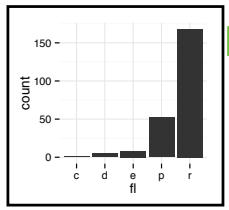
r + theme_classic()

White background no gridlines



r + theme_grey()

Grey background (default theme)



r + theme_minimal()

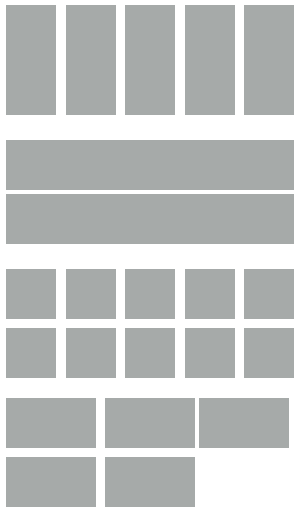
Minimal theme

ggthemes - Package with additional ggplot2 themes

Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables

```
t <- ggplot(mpg, aes(cty, hwy)) + geom_point()
```



t + facet_grid(. ~ fl)
facet into columns based on fl

t + facet_grid(year ~ .)
facet into rows based on year

t + facet_grid(year ~ fl)
facet into both rows and columns

t + facet_wrap(~ fl)
wrap facets into a rectangular layout

Set **scales** to let axis limits vary across facets

t + facet_grid(y ~ x, scales = "free")

x and y axis limits adjust to individual facets

- **"free_x"** - x axis limits adjust
- **"free_y"** - y axis limits adjust

Patchwork

patchwork makes it easy to combine separate ggplots into the same graphic, just add plots together.

patchwork can be installed from Github with:

```
# install.packages("devtools")
devtools::install_github("thomasp85/patchwork")
```

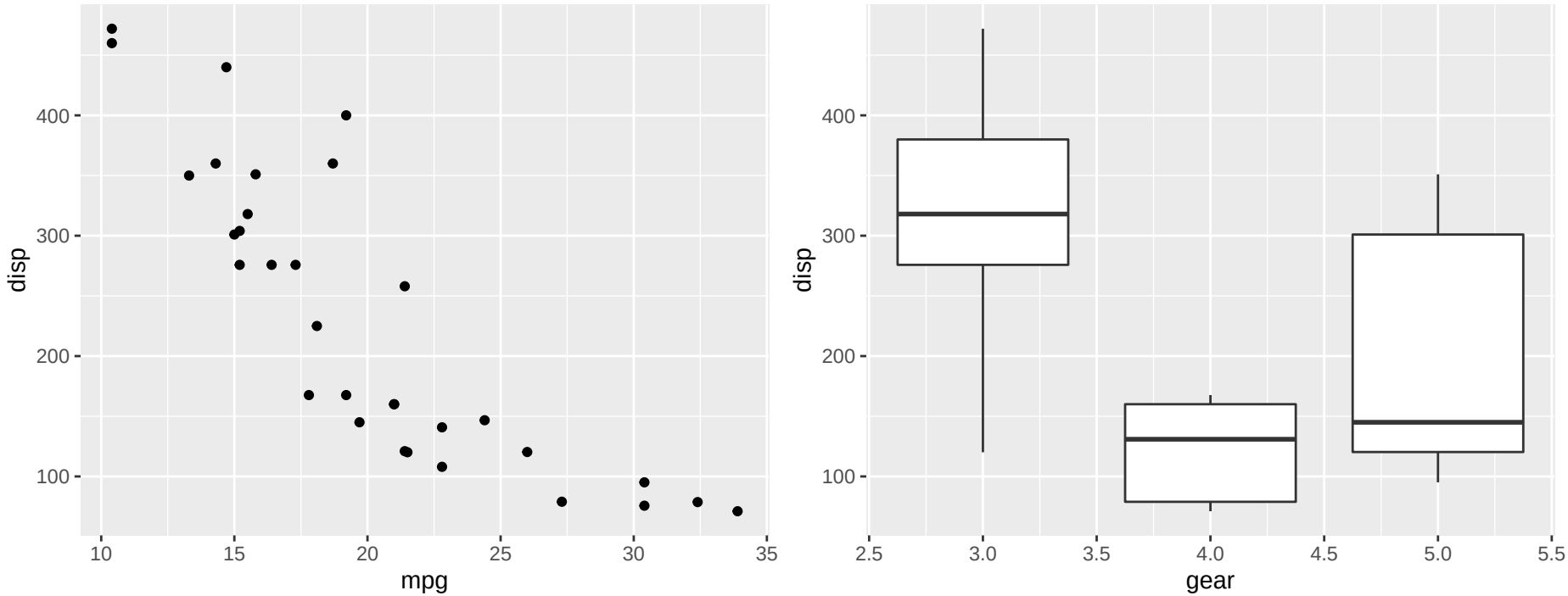
Create two plots

```
library(ggplot2); library(patchwork)

p1 <- ggplot(mtcars) + geom_point(aes(mpg, disp))
p2 <- ggplot(mtcars) + geom_boxplot(aes(gear, disp, group = gea
```

Patchwork

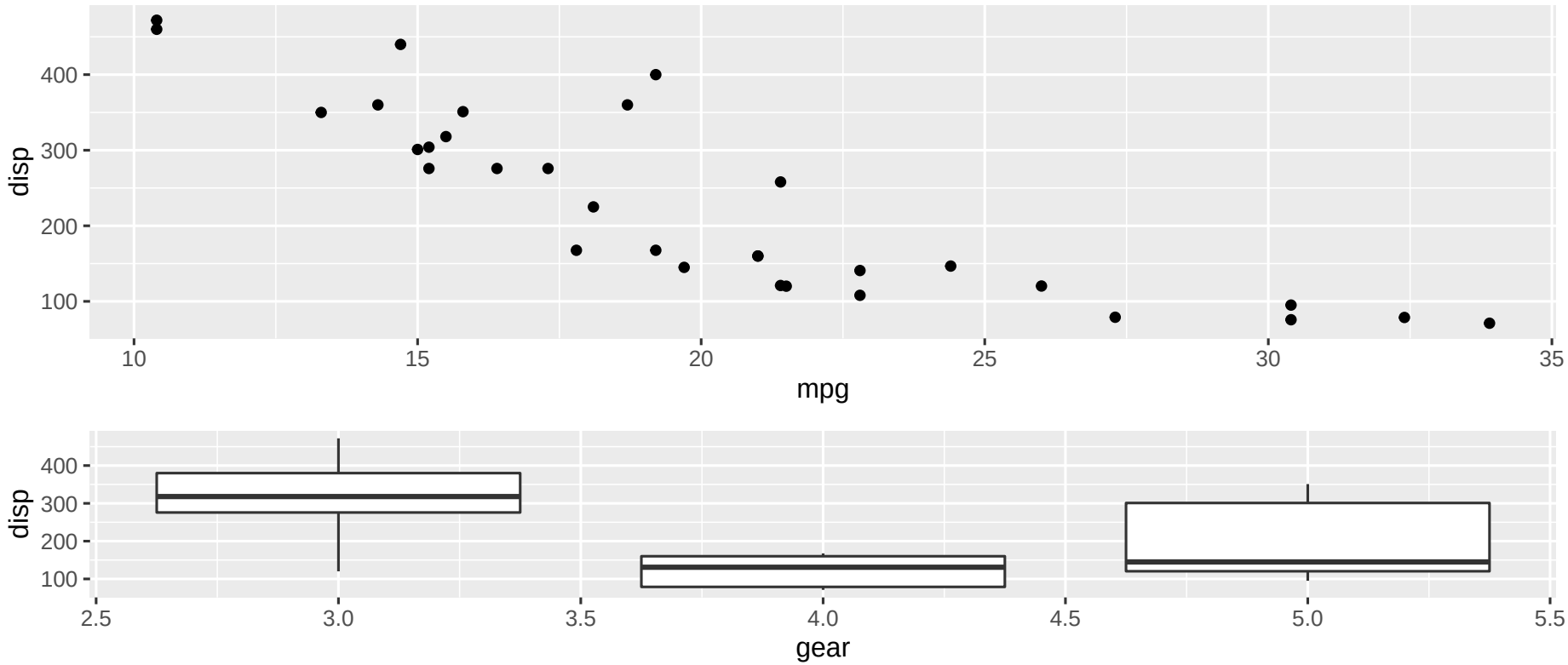
Now, we can simply add the two plots together:

```
p1 + p2
```



plot_layout() lets you define the dimensions of the grid:

```
p1 + p2 + plot_layout(ncol = 1, heights = c(2, 1))
```



For more feature, check out: https://github.com/thomasp85/patchwork

Interactive graphics - plotly

plotly provides an interface to create interactive web graphics via plotly.js (https://plot.ly/).

ggplot2 figures can be made interactive, using the function **ggplotly**:

```
# Load plotly library
library(plotly)

# Create a ggplot object
ggiris <- ggplot() +
  geom_point(data = iris,
             aes(x=Petal.Width, y=Sepal.Length,
                 color = Species))

# Turn plot into an interactive web element
ggplotly(ggiris)
```

Why you should use ggplot2

ggplot2 can be easily combined with **tidyr** and **dplyr**

There is a big community and constant development of new features:

- **ggpmisc** - Add labels of p-value, R2 or equation
- **ggsignif** - Add significance brackets to boxplot
- **ggeffects** - Marginal Effects with ggplot2
- **ggsci** - For scientific journal colour palettes
- **ggrepel** - Avoid overlapping plot labels
- **ggridges** - Create ridgeline plots
- **scatterpie** - Map with embedded pie charts
- **ggtree** - Visualization and annotation of phylogenetic trees
- **ggspatial** - Plotting spatial objects

Do it yourself - Exercise 6

- Plot barchart of the number of species per class and of the number of occurrences for the 10 least common species
- Plot species richness against latitude using different geoms
- Use faceting to split the previous plots by class
- Use patchwork to create individual plots for each class
- Adjust scale, coordinate system, labels and theme
- Plot boxplot of species richness against class and add significant differences (**ggsignif**)
- Make nice map of species richness
- Play around with the data and be creative

Data analysis with dplyr

Do anything

You can use **do()** to perform arbitrary computation, returning either a data frame or arbitrary objects which will be stored in a list. This is particularly useful when working with models.

You can fit models per group with **do()** and then flexibly extract components with either another **do()** or **summarise()**:

```
# Run linear model for each group
library(dplyr)
models <- mtcars %>% group_by(cyl) %>%
  do(mod = lm(mpg ~ disp, data = .))

# Extract coefficients of linear model
models %>% do(data.frame(var = names(coef(.$mod)),
                         coef(summary(.$mod))))
```

ANOVA

do() also works with ANOVAs. But, we need to convert the output into a tidy object using the **tidy** function of the **broom** package:

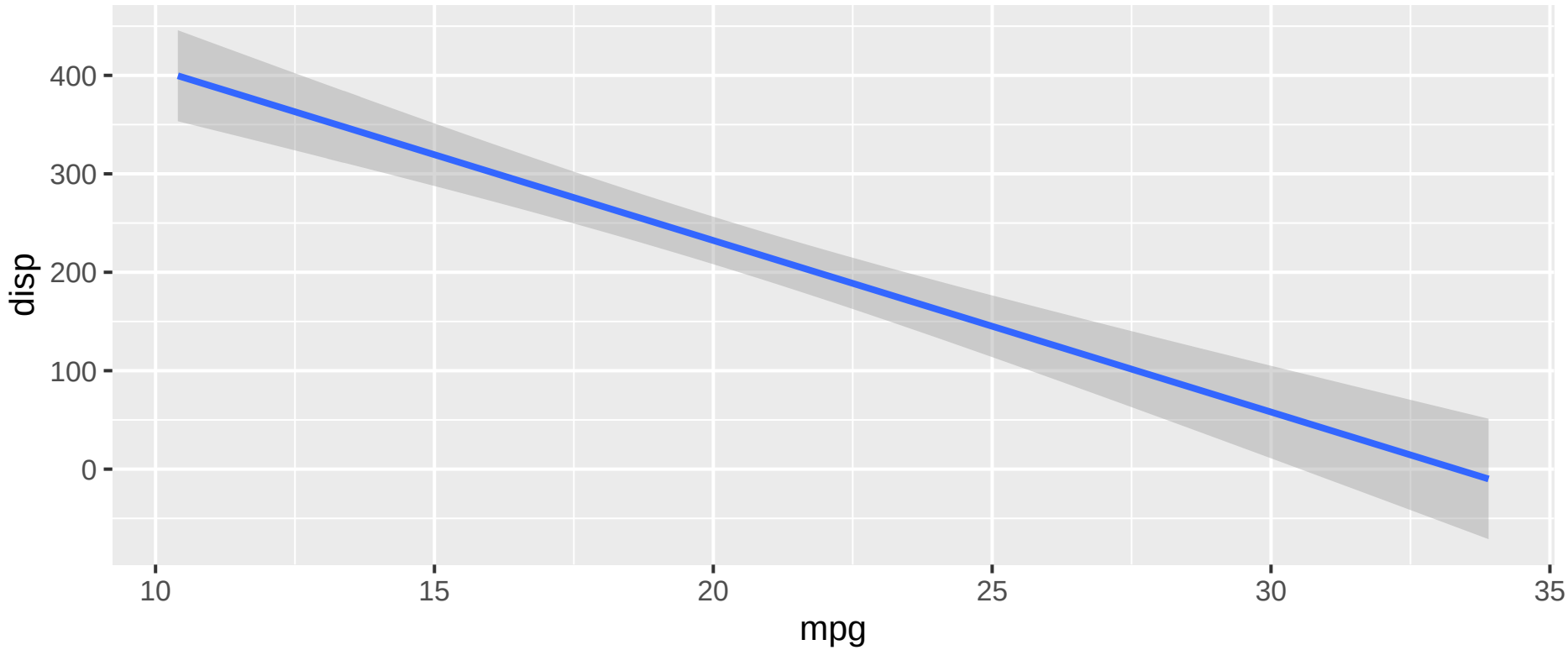
```
# A simply ANOVA
mtcars %>% do(broom::tidy(aov(mpg ~ vs * am * gear, data=.)))

# ANOVA of two linear models for 3 cylinder classes
mtcars %>% group_by(cyl) %>% do(
  mod_linear = lm(mpg ~ disp, data = .),
  mod_quad = lm(mpg ~ poly(disp, 2), data = .)) %>%
  do(aov = anova(.$mod_linear, .$mod_quad)) %>%
  rowwise %>% do(tidy(.$aov)) %>% tidyr::drop_na()
```

broom includes three functions, **tidy(), augment() and glance()**, which all return different model outputs as a data.frame.

Models with ggplot2

You can use **geom_smooth()** to create models and plot them:

```
mtcars %>% ggplot(aes(mpg, disp)) + geom_smooth(method="lm")
```

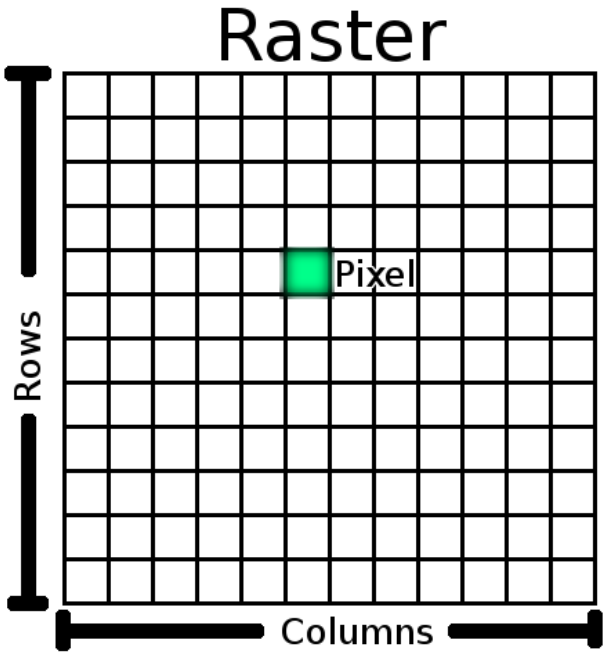


Do it yourself - Exercise 7

- Create linear model using **do()**, i.e. of species richness against latitude for each class
- Run and plot **lm** and **gam** using ggplot2, i.e. of species richness against latitude for each class
- Add equation of lm to plot using **ggpmisc**
- Run ANOVA, i.e. compare linear models of the different classes

raster data with stars

stars - Read raster data



Raster data can be read with the **stars** or alternatively with the **raster** package:

```
library(stars)
tif <- system.file("tif/L7_ETMs.tif",
                   package = "stars")
tif <- read_stars(tif)
```

Raster data can be easily converted into a data.frame and then analysed using **tidyr** and **dplyr**:

```
df <- as.data.frame(tif)
```

vector data with sf

sf - Read vector data

Vector data can be point, line or polygon data:

Vector data can be read with the **sf** package or alternatively with the **rgdal** package:

```
library(sf)
nc <- system.file("shape/nc.shp", package = "sf")
nc <- sf::st_read(nc)
```

And can be plotted with ggplot2 using **geom_sf()**:

```
nc %>% ggplot() + geom_sf()
```

Do it yourself - Exercise 8

- Install the **stars** and **sf** package
- Read tas_ewembi_deu_1981_2010.nc data into R using **read_ncdf()**
- Create date column and split date into day, month and year
- Create polar plot of mean daily temperature split by years
- Plot map of 30-year average temperature, read gadm36_DEU_adm1.shp into R and add polygon of Germany to map
- Calculate monthly mean temperature across years, calculate a linear model for each grid cell, plot map of slope and R2 and add scale bar using **ggspatial**
- Join 30-year average temperature with species data and look at the relationship between richness and temperature per class

Thank you for your attention!