Data Wrangling with *dplyr* & *tidyr*
An introduction

Pierre-Alexandre Fonta

R Lunchs - UniGe

10.04.18

Disclaimer

This presentation is a voluntary work.

My current and past institutions are therefore not responsible for its content.

Objectives

Discover how to easily go from raw data to data ready for analytics with R.

Make your life easier with *dplyr* & *tidyr*.

Increase Data Literacy.

Definitions

Data Analytics

Data Analytics is the discovery, interpretation, and communication of meaningful patterns in data.

https://en.wikipedia.org/wiki/Analytics

Data Wrangling

Data Wrangling [...] is the process of transforming and mapping data from one 'raw' data form into another format with the intent of making it <u>more appropriate and valuable</u> for a variety of downstream purposes such as analytics.

https://en.wikipedia.org/wiki/Data_wrangling

Data Literacy

Data Literacy is the ability to read, understand, create and communicate data as information. [...] [D]ata literacy focuses on the competencies involved in working with data.

As data collection and sharing become routine and data analysis [...] become[s] common ideas in the news, business, government and society, it becomes more and more important for students, citizens, and readers to have some data literacy.

https://en.wikipedia.org/wiki/Analytics

From the field

Personal experience - Various data

numbers + texts + networks

- bibliographic databases (Web of Science, PubMed)

numbers

- serious game (RTS Tabula Rasa)
- economic data (IHS Life Sciences)
- operations data (company)
- statistical data (INSEE)

numbers + networks

- social networks (Twitter)

states / events sequences

- biographical longitudinal data (TraMineR)

numbers + texts

- semantic text annotations

texts

- press releases
- web pages / blogs
- CVs (internal)
- job boards (Indeed)

Personal experience - Various environments

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Blue
Brain
Project

UNIVERSITÉ
DE GENÈVE

LIVES
Pôle de recherche national

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

h e g
Haute école de gestion
Genève

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

randstad

Tips - The Zen of Python

***Beautiful** is better than ugly.*
***Explicit** is better than implicit.*
***Simple** is better than complex.*
***Flat** is better than nested.*
***Readability** counts.*

Special cases aren't special enough to break the rules.
Although practicality beats purity.
In the face of ambiguity, refuse the temptation to guess.
There should be one - and preferably only one - obvious way to do it.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

https://www.python.org/dev/peps/pep-0020/#id3

Tips - Might be obvious, but… 1/2

We need to understand the logic tomorrow. Our colleagues too.

Reusing variables part of the transformation logic is almost ‘irresponsible’.

Modifying a logic used several times but not generalized as a function is a nightmare.

The dataset structure and the data should not be modified by the same logic / code.

Tips - Might be obvious, but... 2/2

The time spend to format the logic to present it to non expert people is significant.

No time for implementing or optimizing the operations at the dataset level.

Data rule. We need interactivity to see them before and after.

We need a ‘notebook’ to capture methodology and results.

tibble

A modern *data.frame*!

At the beginning of the tidyverse were...

Tibbles!

Comparing to data.frames, they:

- complain when a variable does not exist
- don’t change variable names or types
- don’t do partial matching
- have a better print()

tidyr

Create tidy data!

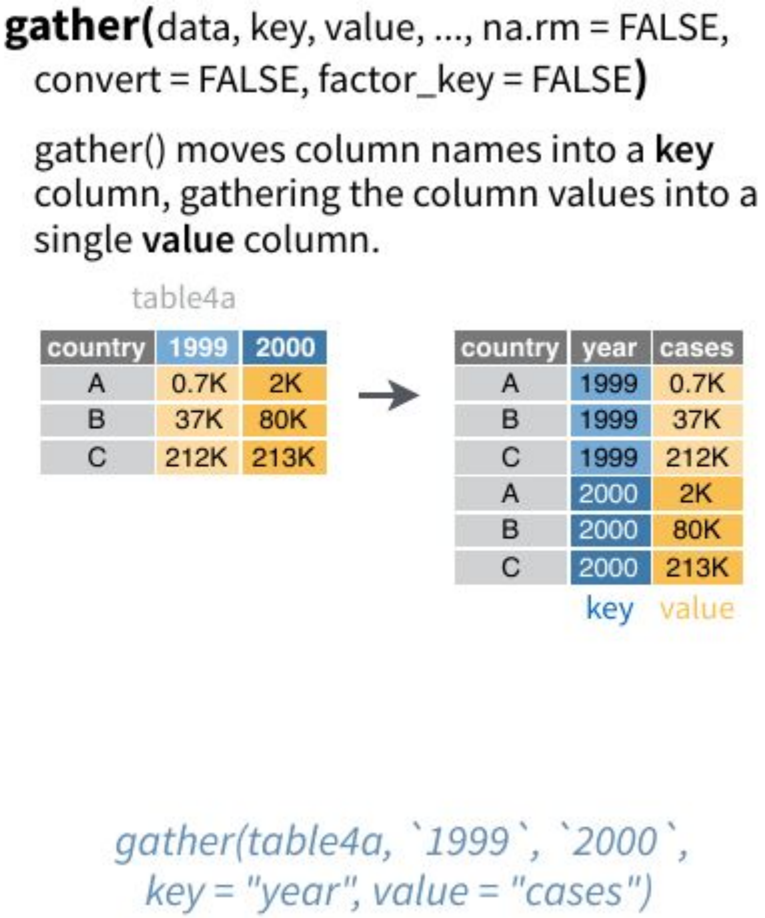
Tidy data

Each variable is in a **column**.

Each observation is a **row**.

Each value is a **cell**.

tidyr - go from wide to long



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

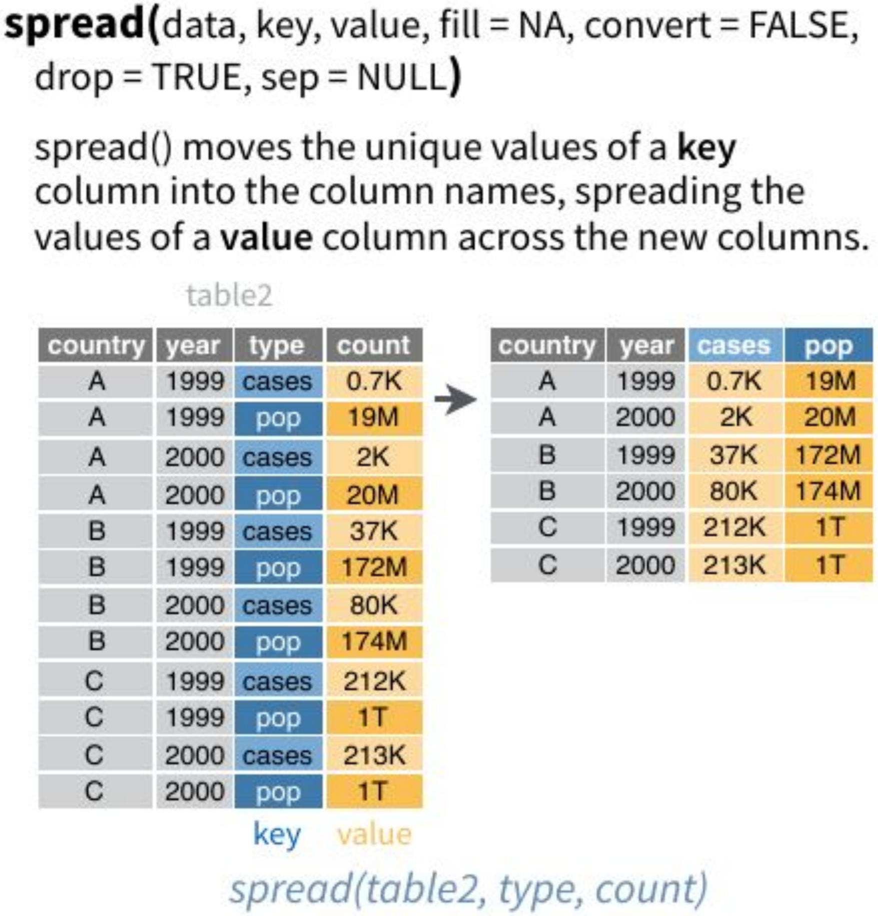
country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

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

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

tidyr - go from long to wide



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.

table2

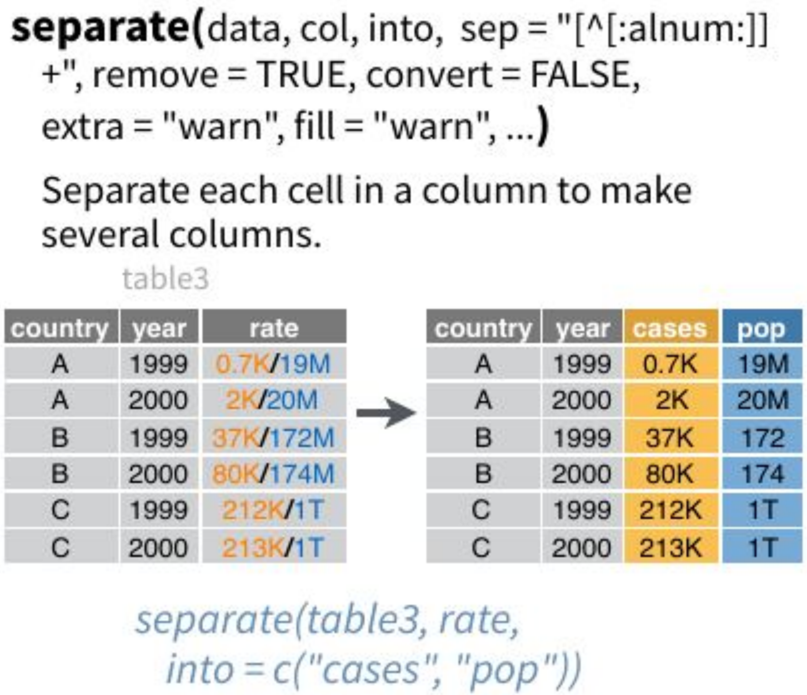
country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T
		key	value

→

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 - split a column into columns



```
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

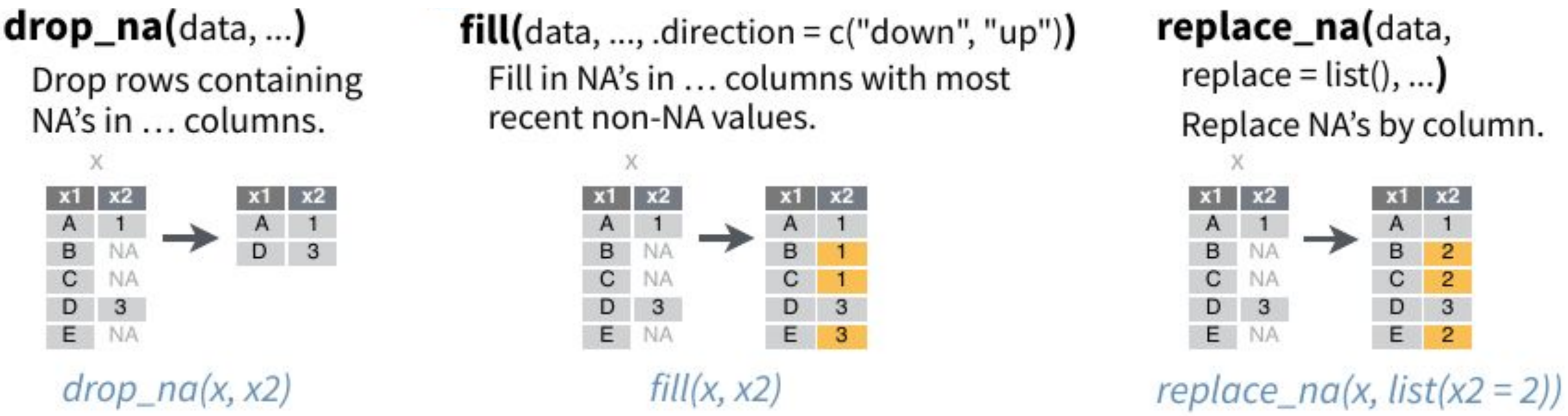
country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

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

tidyr - 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))

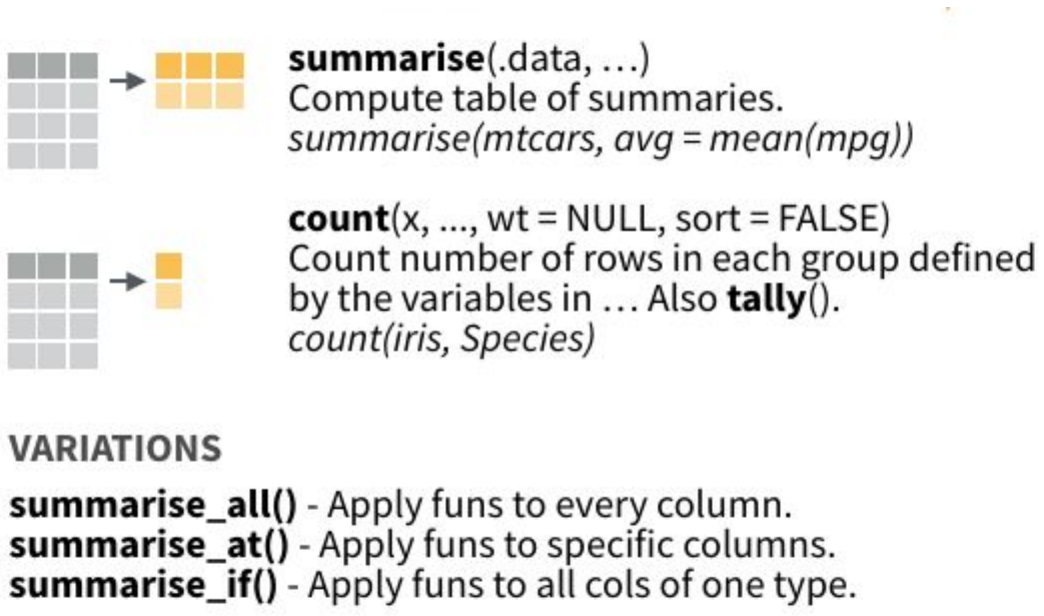
dplyr

Handle your data!

Pipes

x %>% f(y) is equivalent to f(x, y)

dplyr - summarise rows



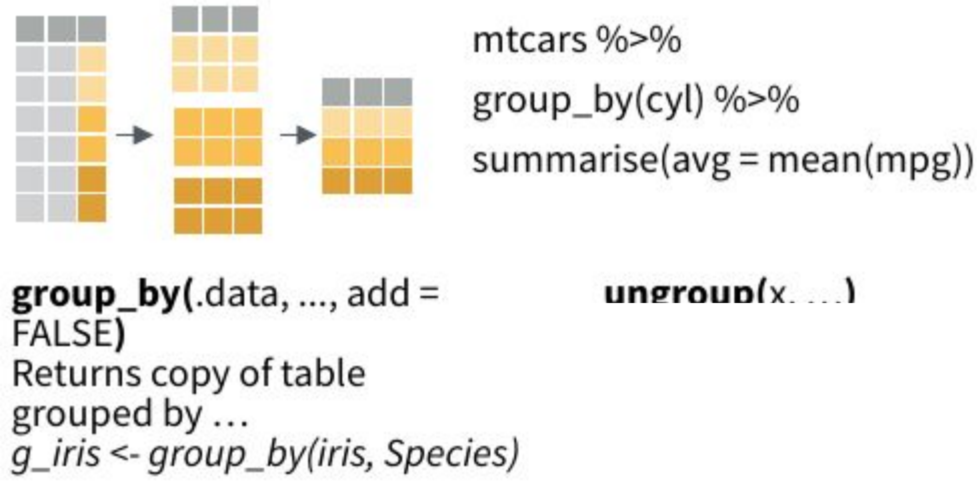
summarise(.data, …)
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

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 - group rows by variable(s)

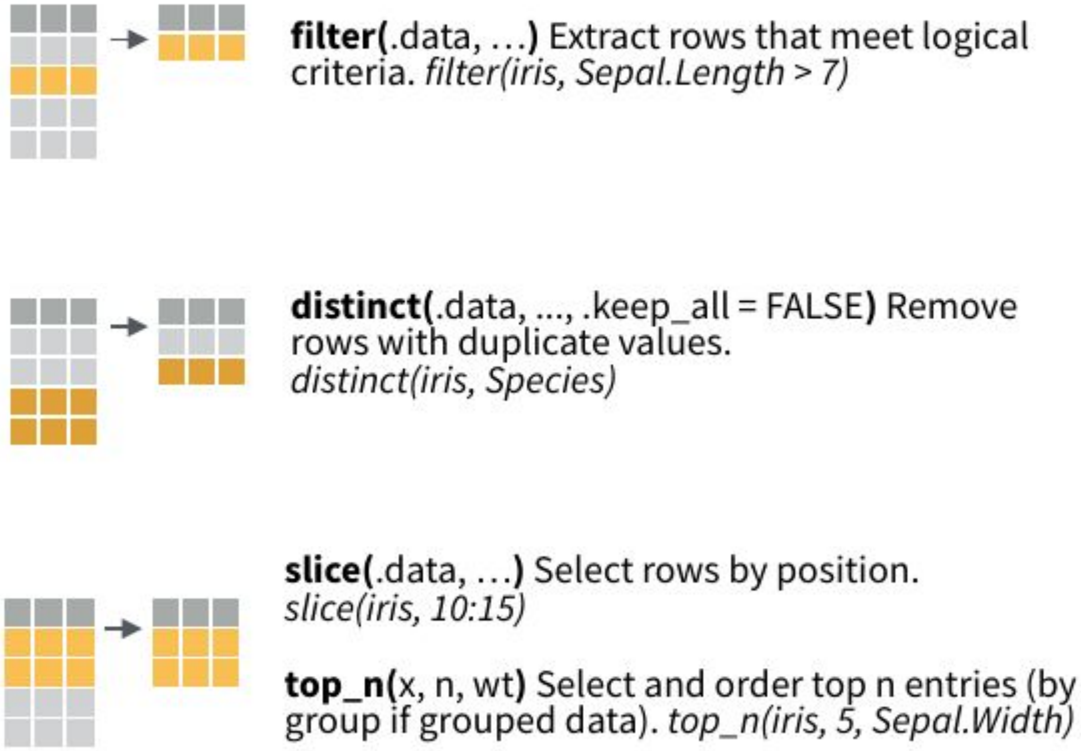


```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

group_by(.data, ..., add = FALSE**)**
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

ungroup(x, …**)**

dplyr - keep specific rows



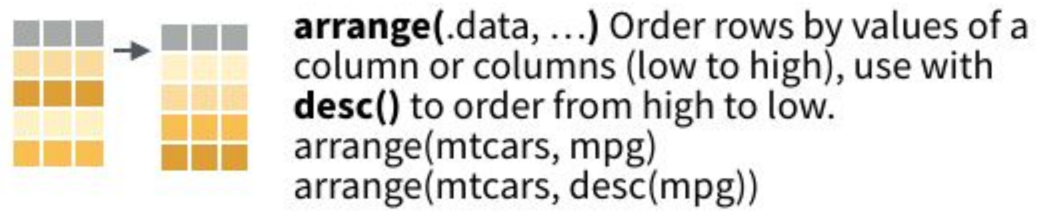
filter(.data, …**)** Extract rows that meet logical criteria. *filter(iris, Sepal.Length > 7)*

distinct(.data, ..., .keep_all = FALSE**)** Remove rows with duplicate values.
distinct(iris, Species)

slice(.data, …**)** Select rows by position.
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)*

dplyr - order rows

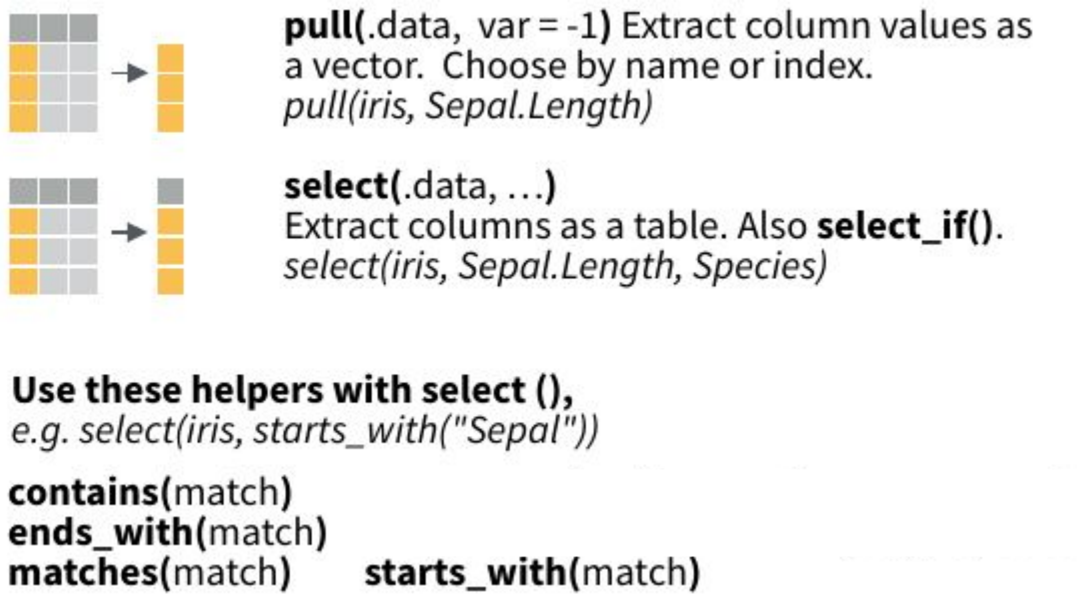


arrange(.data, …**)** Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.

arrange(mtcars, mpg)

arrange(mtcars, desc(mpg))

dplyr - keep specific variables



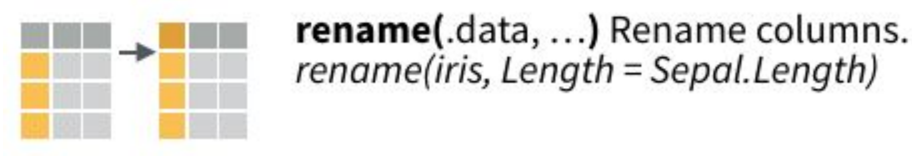
pull(.data, var = -1**)** Extract column values as a vector. Choose by name or index.
pull(iris, Sepal.Length)

select(.data, …**)**
Extract columns as a table. Also **select_if()**.
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

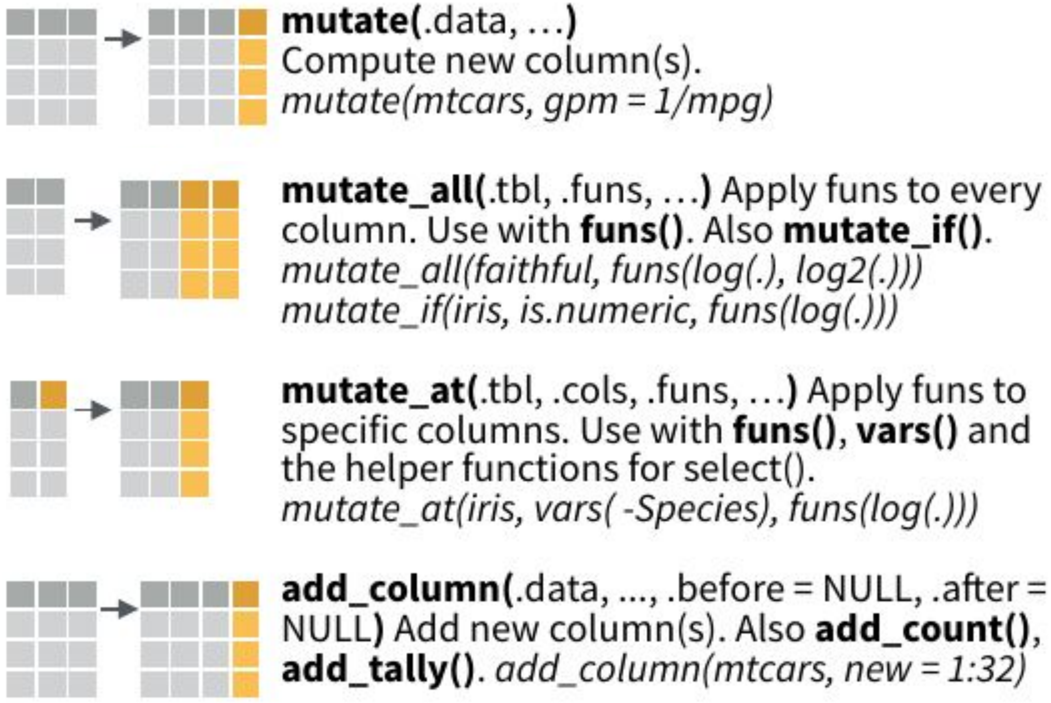
contains(match**)**
ends_with(match**)**
matches(match**)**   **starts_with(**match**)**

dplyr - rename variable



rename(.data, …**)** Rename columns.
rename(iris, Length = Sepal.Length)

dplyr - add new variable(s)



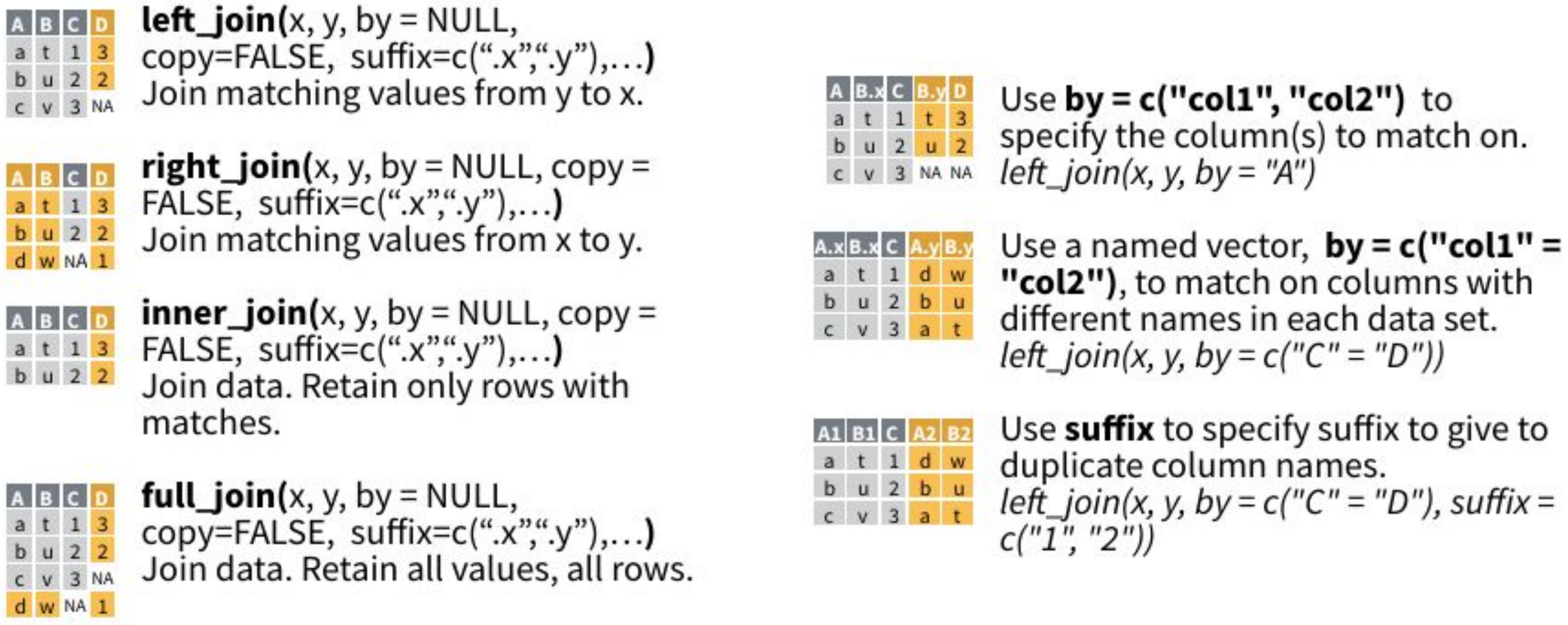
mutate(.data, …**)**
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

mutate_all(.tbl, .funs, …**)** Apply funs to every column. Use with **funs()**. Also **mutate_if()**.
mutate_all(faithful, funs(log(.), log2(.)))
mutate_if(iris, is.numeric, funs(log(.)))

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(.)))

add_column(.data, ..., .before = NULL, .after = NULL**)** Add new column(s). Also **add_count()**, **add_tally()**. *add_column(mtcars, new = 1:32)*

dplyr - join datasets



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.

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.

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")

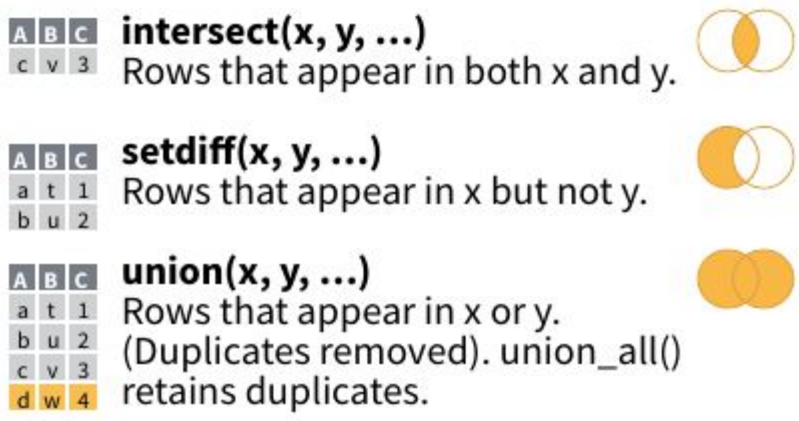
A.x	B.x	C	A.y	B.y
a	t	1	d	w
b	u	2	b	u
c	v	3	a	t

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"))

A1	B1	C	A2	B2
a	t	1	d	w
b	u	2	b	u
c	v	3	a	t

Use **suffix** to specify suffix to give to duplicate column names.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

dplyr - set operations on rows



A	B	C
c	v	3

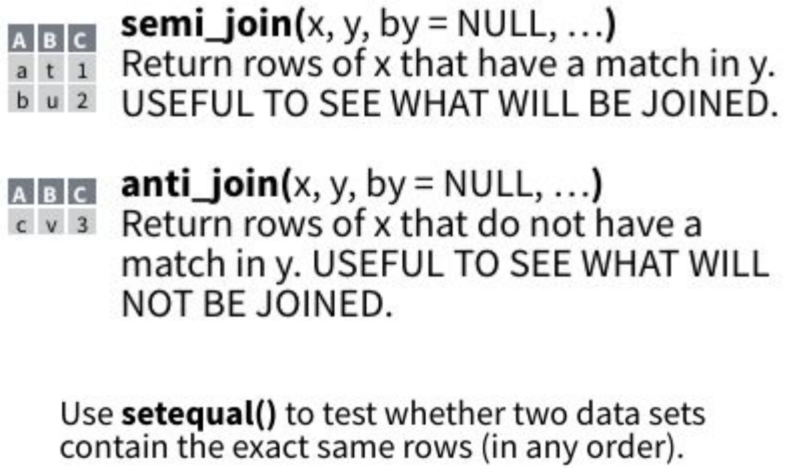
intersect(x, y, ...)
Rows that appear in both x and y.

A	B	C
a	t	1
b	u	2

setdiff(x, y, ...)
Rows that appear in x but not y.

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

union(x, y, ...)
Rows that appear in x or y.
(Duplicates removed). union_all()
retains duplicates.



A	B	C
a	t	1
b	u	2

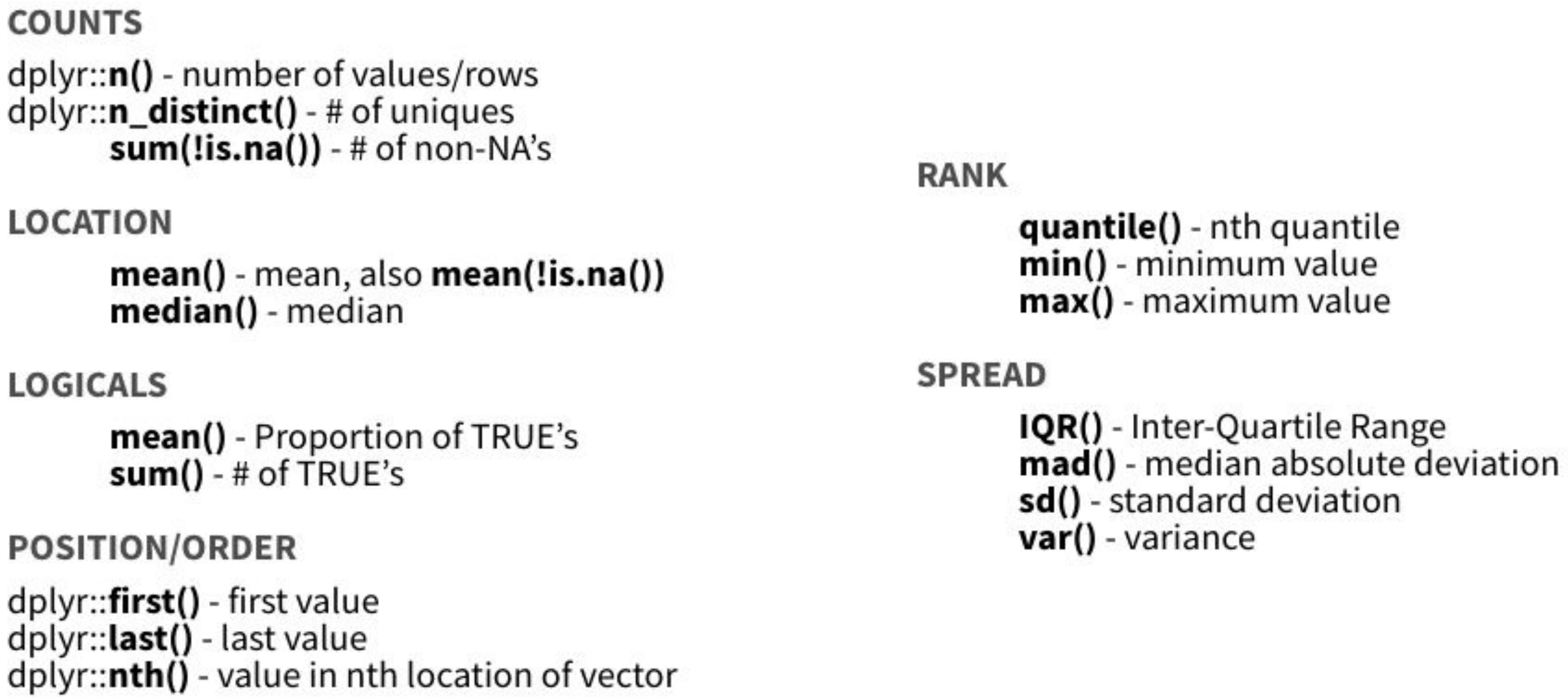
semi_join(x, y, by = NULL, …**)**
Return rows of x that have a match in y.
USEFUL TO SEE WHAT WILL BE JOINED.

A	B	C
c	v	3

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.

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

dplyr - functions for summarise()



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

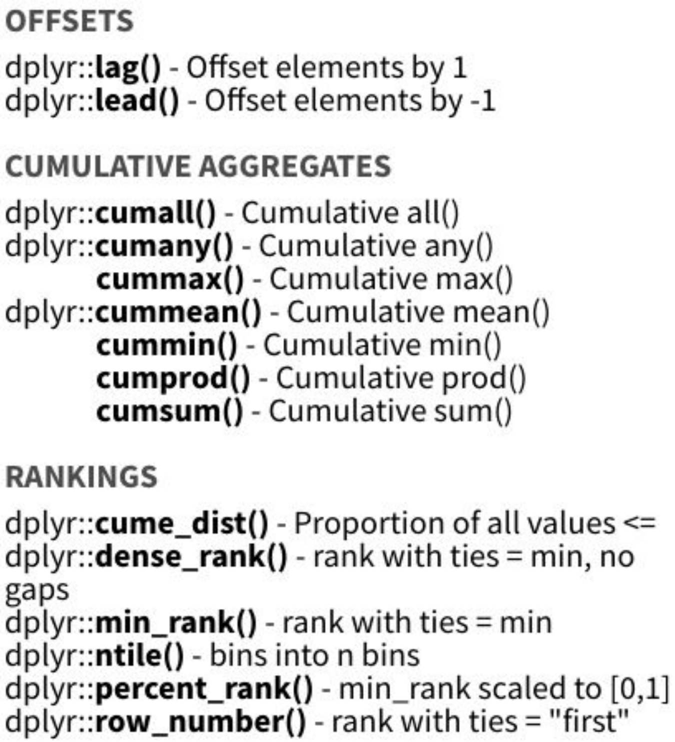
RANK

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

SPREAD

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

dplyr - functions for mutate()



OFFSETS

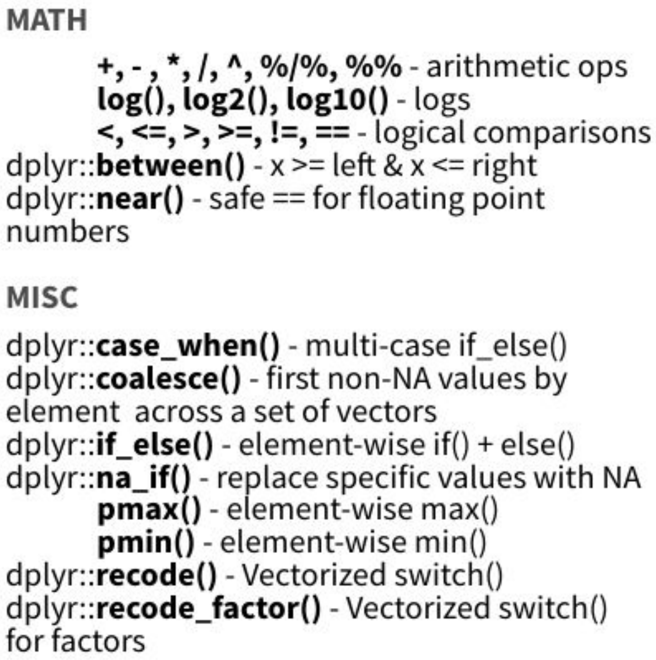
dplyr::**lag()** - Offset elements by 1
dplyr::**lead()** - Offset elements by -1

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
dplyr::**between()** - x >= left & x <= right
dplyr::**near()** - safe == for floating point numbers

MISC

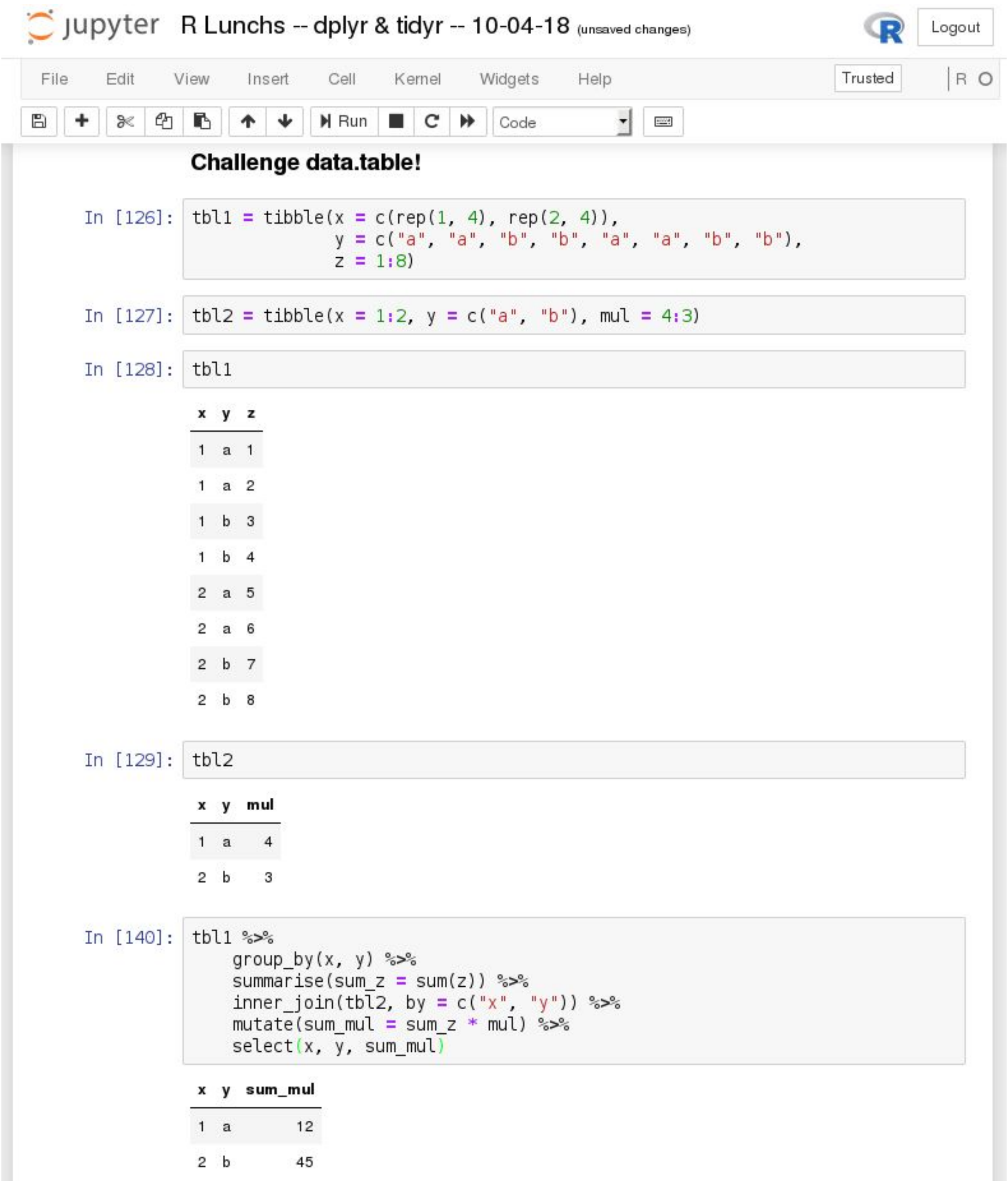
dplyr::**case_when()** - multi-case if_else()
dplyr::**coalesce()** - first non-NA values by element across a set of vectors
dplyr::**if_else()** - element-wise if() + else()
dplyr::**na_if()** - replace specific values with NA
pmax() - element-wise max()
pmin() - element-wise min()
dplyr::**recode()** - Vectorized switch()
dplyr::**recode_factor()** - Vectorized switch() for factors

Jupyter
A multilanguage notebook!

Jupyter Notebook

———

https://jupyter.org



Challenge data.table!

```
In [126]: tbl1 = tibble(x = c(rep(1, 4), rep(2, 4)),
                        y = c("a", "a", "b", "b", "a", "a", "b", "b"),
                        z = 1:8)
```

```
In [127]: tbl2 = tibble(x = 1:2, y = c("a", "b"), mul = 4:3)
```

```
In [128]: tbl1
```

x	y	z
1	a	1
1	a	2
1	b	3
1	b	4
2	a	5
2	a	6
2	b	7
2	b	8

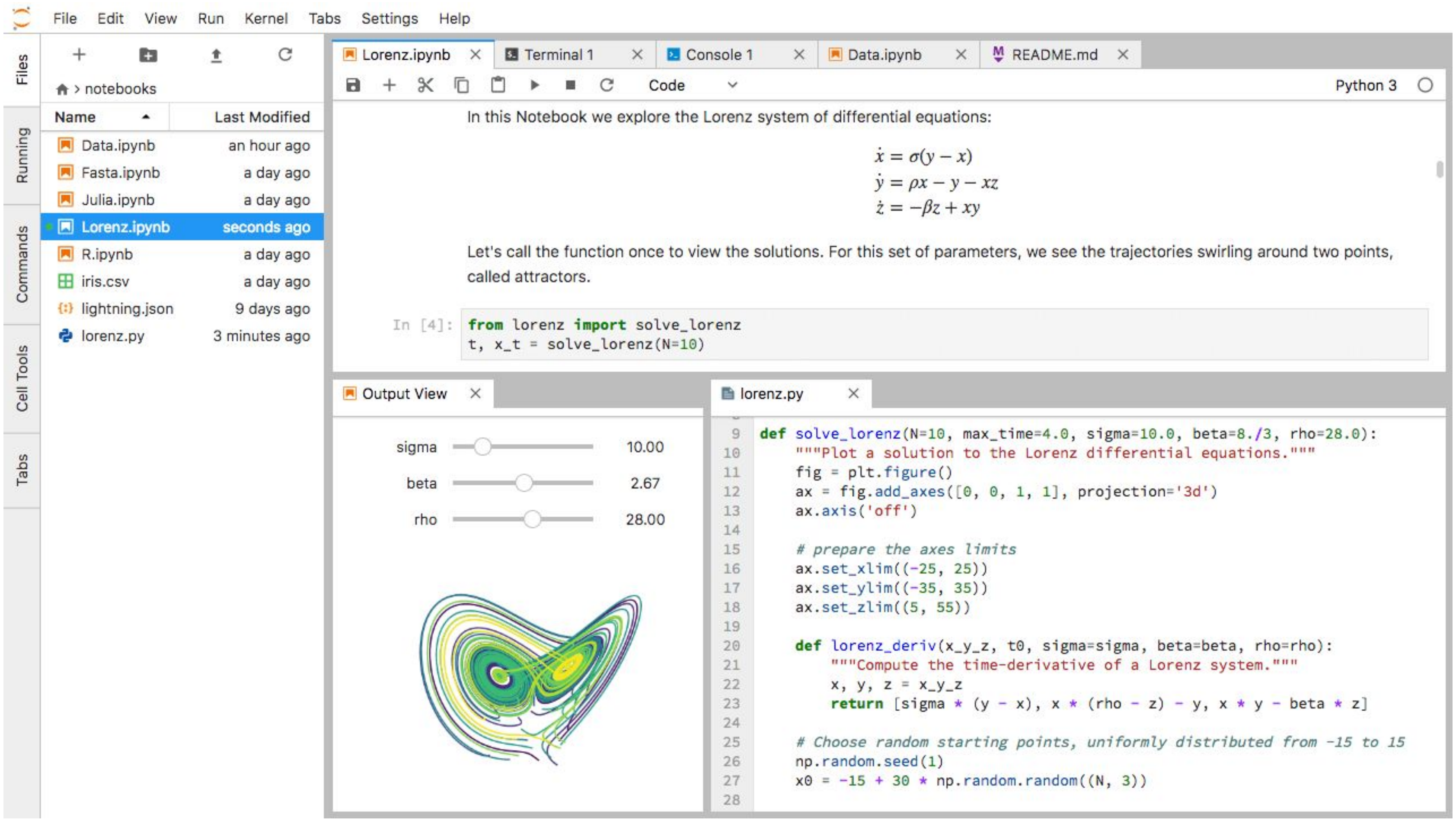
```
In [129]: tbl2
```

x	y	mul
1	a	4
2	b	3

```
In [140]: tbl1 %>%
              group_by(x, y) %>%
              summarise(sum_z = sum(z)) %>%
              inner_join(tbl2, by = c("x", "y")) %>%
              mutate(sum_mul = sum_z * mul) %>%
              select(x, y, sum_mul)
```

x	y	sum_mul
1	a	12
2	b	45

JupyterLab - Coming Soon (beta)



Name	Last Modified
Data.ipynb	an hour ago
Fasta.ipynb	a day ago
Julia.ipynb	a day ago
Lorenz.ipynb	seconds ago
R.ipynb	a day ago
iris.csv	a day ago
lightning.json	9 days ago
lorenz.py	3 minutes ago

In this Notebook we explore the Lorenz system of differential equations:

$$\begin{aligned}\dot{x} &= \sigma(y - x)\\ \dot{y} &= \rho x - y - xz\\ \dot{z} &= -\beta z + xy\end{aligned}$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

```
In [4]: from lorenz import solve_lorenz
        t, x_t = solve_lorenz(N=10)
```

```
sigma  10.00
beta    2.67
rho    28.00
```

```
def solve_lorenz(N=10, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

```

https://jupyterlab.readthedocs.io/en/latest/images/jupyterlab.png

Demo

Takeaways

Takeaways

---

dplyr / *tidyr* syntax

- beautiful (~~ugly~~)
- explicit (~~implicit~~)
- simple (~~complex~~)
- readable, easy to explain

tidy data

- flat (~~nested~~)
- analytics oriented

Jupyter

- interactive
- code, outputs, and comments in one place

To go further

Databases (*dbplyr*)
http://dbplyr.tidyverse.org

Programming
http://dplyr.tidyverse.org/articles/programming.html

Windows functions
http://dplyr.tidyverse.org/articles/window-functions.html

References

---

tibble
http://tibble.tidyverse.org

dplyr
http://dplyr.tidyverse.org
https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf

tidyr
http://tidyr.tidyverse.org
https://github.com/rstudio/cheatsheets/raw/master/data-import.pdf