

31
10
23

essencemedia.com

Wprowadzenie do Tidyverse

Część III



Co dzisiaj

Łączenie danych (join)

Praca z datami (biblioteka lubridate)

Wizualizacja danych (ggplot)

Łączenie danych (join)

Biblioteki w ramach Tidyverse'a

Import
danych



Uporządkowanie
danych



**Przetwarzanie
danych**



Programowanie

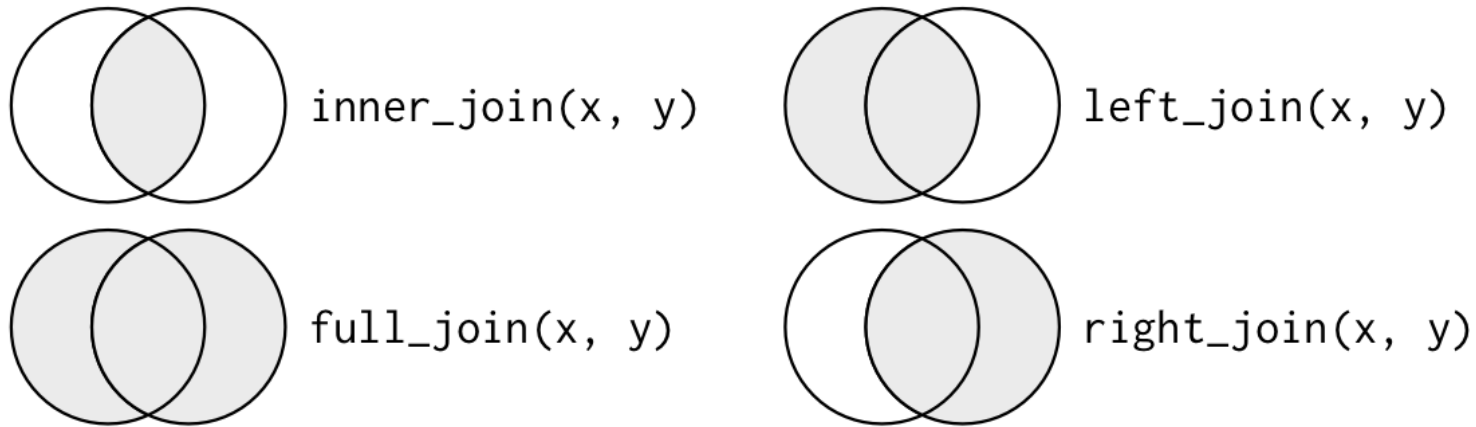


Wizualizacja



Łączenie zbiorów danych za pomocą poleceń **join**

Mutating joins



Jak również *cross join*, *filter joins*, *nest join*

Praca z danymi

Biblioteki w ramach Tidyverse'a

Import
danych



Uporządkowanie
danych



**Przetwarzanie
danych**



Programowanie



Wizualizacja



Praca z datami: lubridate

Funkcja	Jak również	Opis
as_date()	as.Date() [base] as_datetime()	przekształca tekst na datę w formacie systemowym (wymaga odpowiedniego formatu)
today()	now()	podaje bieżącą datę
year()	month() , day() , ...	zwraca odpowiedni fragment daty
years()	months() , days() , ...	służą do operacji na datach
ymd()	dmy() , mdy() , ...	zamienia tekst (w odpowiednim formacie) w datę
floor_date()	ceiling_date() , round_date()	zaokrągla datę w dół do wielokrotności jednostki czasu

Dla spragnionych wiedzy: lubridate cheat sheet

Dates and times with lubridate : : CHEAT SHEET

Date-times

2017-11-28 12:00:00
A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC.
`dt <- as_datetime(1511870400)`
"2017-11-28 12:00:00 UTC"

PARSE DATE-TIMES (Convert strings or numbers to date-times)

1. Identify the order of the year (y), month (m), day (d), hour (h), minute (m) and second (s) elements in your data
2. Use the function below whose name replicates the order. Each accepts a wide variety of input formats.

2017-11-28T14:02:00
`ymd_hms()`, `ymd_hm()`, `ymd_h()`,
`ymd_hms("2017-11-28T14:02:00")`
2017-22-12 10:00:00
`ydym_hms()`, `ydym_hm()`, `ydym_h()`,
`ydym_hms("2017-22-12 10:00:00")`
11/28/2017 1:02:03
`mdy_hms()`, `mdy_hm()`, `mdy_h()`,
`mdy_hms("11/28/2017 1:02:03")`
1 Jan 2017 23:59:59
`dmy_hms()`, `dmy_hm()`, `dmy_h()`,
`dmy_hms("1 Jan 2017 23:59:59")`
20170131
`ymd()`, `ymd()`, `ymd()`, `ymd("20170131")`
July 4th, 2000
`mdy()`, `mdy()`, `mdy()`, `mdy("July 4th, 2000")`
4th of July '99
`dmy()`, `dmy()`, `dmy()`, `dmy("4th of July '99")`
2001 Q3
`yaq()` Q for quarter, `yaq("2001: Q3")`
hms:hms() Also lubridate:hms(),
hm() and ms(), which return
periods. `hms(sec = 0, min = 1,
hours = 2)`
2017.5
`date_decimal(decimal, tz = "UTC")`
Q for quarter, `date_decimal(2017.5)`
`now(tzone = "")` Current time in a tz
(defaults to system tz, `now()`)
`today(tzone = "")` Current date in a tz
(defaults to system tz, `today()`)
`fast_strptime()` Faster strptime,
`fast_strptime("%Y/%m/%d")`
`parse_date_time()` Easier strptime,
`parse_date_time("%Y/%m/%d", "ymd")`

2017-11-28
A **date** is a day stored as the number of days since 1970-01-01
`d <- as_date(17498)`
"2017-11-28"

GET AND SET COMPONENTS

- Use an accessor function to get a component.
Assign into an accessor function to change a component in place.

2018-01-31 11:59:59
`date(x)` Date component, `date(dt)`
`year(x)` Year, `year(dt)`
`isoyear(x)` The ISO 8601 year.
`epiyear(x)` Epidemiological year.
2018-01-31 11:59:59
`month(x, label, abbr)` Month, `month(dt)`
2018-01-31 11:59:59
`day(x)` Day of month, `day(dt)`
`wday(x, label, abbr)` Day of week, `wday(dt)`
`hour(x)` Hour, `hour(dt)`
2018-01-31 11:59:59
`minute(x)` Minutes, `minute(dt)`
2018-01-31 11:59:59
`second(x)` Seconds, `second(dt)`
`week(x)` Week of the year, `week(dt)`
`isoweek()` ISO 8601 week.
`epiweek()` Epidemiological week.
`quarter(x, with_year = FALSE)`
Quarter, `quarter(dt)`
`semester(x, with_year = FALSE)`
Semester, `semester(dt)`
`am(x)` Is it in the am? `am(dt)`
`pm(x)` Is it in the pm? `pm(dt)`
`dst(x)` Is it daylight savings? `dst(dt)`
`leap_year(x)` Is it a leap year?
`leap_year(dt)`
`update(object, ..., simple = FALSE)`
`update(dt, mday = 2, hour = 1)`



Round Date-times

`floor_date(x, unit = "second")`
Round down to nearest unit.
`floor_date(dt, unit = "month")`
`round_date(x, unit = "second")`
Round to nearest unit.
`round_date(dt, unit = "month")`
`ceiling_date(x, unit = "second")`
Change on boundary = NULL.
Round up to nearest unit.
`ceiling_date(dt, unit = "month")`
`rollback(dates, roll_to = first = FALSE, preserve_hms = TRUE)`
Roll back to last day of previous month, `rollback(dt)`

Stamp Date-times

`stamp()` Derive a template from an example string and return a new function that will apply the template to date-times. Also `stamp_date()` and `stamp_time()`.

1. Derive a template, create a function
`sf <- stamp("Created Sunday, Jan 17, 1999 3:34")`
2. Apply the template to dates
`sfymd("2010-04-05")`
[1] "Created Monday, Apr 05, 2010 00:00"

Tip: use a date with day > 12

Time Zones

R recognizes ~60 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns one time zone per vector.

Use the UTC time zone to avoid Daylight Savings.

`OlsonNames()` Returns a list of valid time zone names. `OlsonNames()`

4:00 Pacific
5:00 Mountain
6:00 Central
7:00 Eastern
7:00 Pacific
7:00 Mountain
7:00 Central
7:00 Eastern
with `tz(time, tzone = "")` Get the same date-time in a new time zone (a new clock time).
with `tz(dt, "US/Pacific")`
force `tz(time, tzone = "")` Get the same clock time in a new time zone (a new date-time).
force `tz(dt, "US/Pacific")`



Math with Date-times

Lubridate provides three classes of timespans to facilitate math with dates and date-times.

Math with date-times relies on the timeline, which behaves inconsistently. Consider how the timeline behaves during:

A normal day
`nor <- ymd_hms("2018-01-01 01:30:00", tz = "US/Eastern")`

The start of daylight savings (spring forward)
`gap <- ymd_hms("2018-03-11 01:30:00", tz = "US/Eastern")`

The end of daylight savings (fall back)
`lap <- ymd_hms("2018-11-04 00:30:00", tz = "US/Eastern")`

Leap years and leap seconds
`leap <- ymd("2019-03-01")`

Periods track changes in clock times, which ignore time line irregularities.

`normal + minutes(90)`

`gap + minutes(90)`

`lap + minutes(90)`

`leap + years(1)`

Durations track the passage of physical time, which deviates from clock time when irregularities occur.

`normal + dminutes(90)`

`gap + dminutes(90)`

`lap + dminutes(90)`

`leap + dyears(1)`

Intervals represent specific intervals of the timeline, bounded by start and end date-times.

`interval(normal, normal + minutes(90))`

`interval(gap, gap + minutes(90))`

`interval(lap, lap + minutes(90))`

`interval(leap, leap + years(1))`

Not all years are 365 days due to **leap days**.
Not all minutes are 60 seconds due to **leap seconds**.

It is possible to create an imaginary date by adding **months**, e.g. February 31st
`jan31 <- ymd(20180131)`
`jan31 + months(1)`
NA

`%m+%` and `%m-%` will roll imaginary dates to the last day of the previous month.
`jan31 %m+ months(1)`
"2018-02-28"

`add_with_rollback(e1, e2, roll_to = first = TRUE)` will roll imaginary dates to the first day of the new month.
`add_with_rollback(jan31, months(1), roll_to = first = TRUE)`
"2018-03-01"



PERIODS

Add or subtract periods to model events that happen at specific clock times, like the NYSE opening bell.

Make a period with the name of a time unit **pluralized**, e.g.

`p <- months(3) + days(12)`
`"3m 12d 0h 0m 0s"`

`years(x = 1)` x years.
`months(x = 1)` x months.
`weeks(x = 1)` x weeks.
`days(x = 1)` x days.
`hours(x = 1)` x hours.
`minutes(x = 1)` x minutes.
`seconds(x = 1)` x seconds.
`milliseconds(x = 1)` x milliseconds.
`microseconds(x = 1)` x microseconds.
`nanoseconds(x = 1)` x nanoseconds.
`picoseconds(x = 1)` x picoseconds.

`period(num = NULL, units = "second", ...)`
An automation friendly period constructor.
`period(5, unit = "years")`

`as.period(x, unit)` Coerce a timespan to a period, optionally in the specified units.
Also `is.period()`, `as.period()`

`period_to_seconds(x)` Convert a period to the "standard" number of seconds implied by the period. Also `seconds_to_period()`, `period_to_seconds(p)`

DURATIONS

Add or subtract durations to model physical processes, like battery life. Durations are stored as seconds, the only time unit with a consistent length.

Difftime are a class of durations found in base R.

Make a duration with the name of a period prefixed with a **d**, e.g.

`dd <- ddays(14)`
`"1209600s (~2 weeks)"`

`years(x = 1)` 31536000x seconds.
`weeks(x = 1)` 604800x seconds.
`ddays(x = 1)` 86400x seconds.
`dhours(x = 1)` 3600x seconds.
`dminutes(x = 1)` 60x seconds.
`dseconds(x = 1)` 1x seconds.
`dmilliseconds(x = 1)` 1x 10⁻³ seconds.
`dmicroseconds(x = 1)` 1x 10⁻⁶ seconds.
`dnanoseconds(x = 1)` 1x 10⁻⁹ seconds.
`dpicoseconds(x = 1)` 1x 10⁻¹² seconds.

`duration(num = NULL, units = "second", ...)`
An automation friendly duration constructor.
`duration(5, unit = "years")`

`as.duration(x, ...)` Coerce a timespan to a duration. Also `is.duration()`, `is.difftime()`, `as.duration()`

`make_difftime(x, start, ...)` Make the intervals that occur between the start date-time and the specified number of units.
Also `diff_time()`, `make_difftime()`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_flip(int)` Reverse the direction of an interval. Also `int_standardize()`, `int_flip()`

`int_shift(int, by)` Shifts an interval up or down the timeline by a timespan. `int_shift(days(1))`

`as.interval(x, start, ...)` Coerce a timespan to an interval. Also `is.interval()`, `is.difftime()`, `as.interval()`

`int_length(int)` Length in seconds. `int_length(i)`

`int_standardize(i)` Standardize the direction of an interval. Also `int_standardize()`, `int_flip()`

Wizualizacja danych: ggplot

Biblioteki w ramach Tidyverse'a

Import
danych



Uporządkowanie
danych



Przetwarzanie
danych



Programowanie



Wizualizacja



Użycie ggplota

Struktura polecenia dla **ggplota**

ggplot (data = <DATA>) +

**<GEOM_FUNCTION> (mapping = aes(<MAPPINGS>),
stat = <STAT> , position = <POSITION>) +**

<COORDINATE_FUNCTION> +

<FACET_FUNCTION> +

<SCALE_FUNCTION> +

<THEME_FUNCTION>

Wywołaj funkcję i podaj data frame, na którym powinna pracować

Podaj wymagane parametry

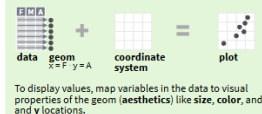
Dodatkowe argumenty: wygląd, skala, podział na pod-wykresy itd.

ggplot2 cheat sheet

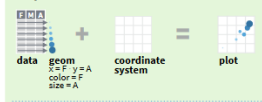
Data visualization with ggplot2 : : CHEAT SHEET

Basics

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



To display values, map variables in the data to visual properties of the geom (aesthetics) like size, color, and x and y locations.



Complete the template below to build a graph.

```
ggplot(data = <DATA>) +  
  <GEOM_FUNCTION>(<mapping> = aes(<MAPPINGS>)) +  
  <STAT_FUNCTION>(<position> = <POSITION>) +  
  <COORDINATE_FUNCTION> +  
  <SCALE_FUNCTION> +  
  <THEME_FUNCTION>
```

ggplot(data = mpg, aes(x = cty, y = hwy)) Begins a plot that you finish by adding layers to. Add one geom function per layer.

last_plot() Returns the last plot.

ggsave("plot.png", width = 5, height = 5) Saves last plot as 5 x 5 file named "plot.png" in working directory. Matches file type to file extension.

Aes

Common aesthetic values. color and fill - string ("red", "#RRGGBB")
linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotteddash", 5 = "longdash", 6 = "twodash")
linewidth - string ("round", "butt", or "square")
linejoin - string ("round", "mitre", or "bevel")
size - integer (line width in mm)
shape - integer/shape name or a single character ("a")



Geoms

Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

GRAPHICAL PRIMITIVES

a <- ggplot(economics, aes(date, unemployment))
b <- ggplot(seals, aes(x = long, y = lat))

a <- geom_blank() and a <- expand_limits() Ensure limits include values across all plots.
b <- geom_curve(aes(yend = lat + 1, xend = long + 1, curvature = 1) - x, yend, yend, alpha, color, color, curvature, linetype, size)

a <- geom_path(linewidth = "butt", linejoin = "round", linemitre = 1) x, y, alpha, color, group, linetype, size

a <- geom_polygon(aes(alpha = 50)) - x, y, alpha, color, fill, group, subgroup, linetype, size

a <- geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1) - x, y, alpha, color, fill, group, linetype, size

a <- geom_ribbon(aes(ymin = unemployment - 900, ymax = unemployment + 900)) - x, y, alpha, color, fill, group, linetype, size

LINE SEGMENTS common aesthetics: x, y, alpha, color, linetype, size

b <- geom_abline(aes(intercept = 0, slope = 1))
b <- geom_hline(aes(yintercept = lat))
b <- geom_vline(aes(xintercept = long))

b <- geom_segment(aes(yend = lat + 1, xend = long + 1))
b <- geom_spoke(aes(angle = 1:1155, radius = 1))

ONE VARIABLE continuous

c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

c <- geom_area(stat = "bin") x, y, alpha, color, fill, linetype, size

c <- geom_density(kernel = "gaussian") x, y, alpha, color, fill, group, linetype, size, weight

c <- geom_dotplot() x, y, alpha, color, fill

c <- geom_freqpoly() x, y, alpha, color, group, linetype, size

c <- geom_histogram(binwidth = 5) x, y, alpha, color, fill, linetype, size, weight

c2 <- geom_qq(aes(sample = hwy)) x, y, alpha, color, fill, linetype, size, weight

discrete

d <- ggplot(mpg, aes(fill))

d <- geom_bar() x, alpha, color, fill, linetype, size, weight

TWO VARIABLES

both continuous
e <- ggplot(mpg, aes(cty, hwy))

e <- geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e <- geom_point() x, y, alpha, color, fill, shape, size, stroke

e <- geom_quantile() x, y, alpha, color, group, linetype, size, weight

e <- geom_rug(sides = "b") x, y, alpha, color, linetype, size

e <- geom_smooth(method = lm) x, y, alpha, color, fill, group, linetype, size, weight

e <- geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous

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

f <- geom_col() x, y, alpha, color, fill, group, linetype, size

f <- geom_boxplot() x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f <- geom_dotplot(binaxis = "y", stackdir = "center") x, y, alpha, color, fill, group

f <- geom_violin(scale = "area") x, y, alpha, color, fill, group, linetype, size, weight

both discrete

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

g <- geom_count() x, y, alpha, color, fill, shape, size, stroke

g <- geom_jitter(height = 2, width = 2) x, y, alpha, color, fill, shape, size

THREE VARIABLES

sealsSz <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))

l <- geom_contour(aes(z)) x, y, z, alpha, color, group, linetype, size, weight

l <- geom_contour_filled(aes(fill = z)) x, y, alpha, color, fill, group, linetype, size, subgroup

continuous bivariate distribution

h <- ggplot(diamonds, aes(carat, price))

h <- geom_bin2d(binwidth = c(0.25, 500)) x, y, alpha, color, fill, linetype, size, weight

h <- geom_density_2d() x, y, alpha, color, group, linetype, size

h <- geom_hex() x, y, alpha, color, fill, size

continuous function

i <- ggplot(economics, aes(date, unemployment))

i <- geom_area() x, y, alpha, color, fill, linetype, size

i <- geom_line() x, y, alpha, color, group, linetype, size

i <- geom_step(direction = "hv") x, y, alpha, color, group, linetype, size

visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)

j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))

j <- geom_crossbar(aes(x = grp, y = fit, ymin = fit - se, ymax = fit + se))

j <- geom_errorbar() - x, y, ymin, ymax, alpha, color, group, linetype, size, width

j <- geom_linerange() x, y, ymin, ymax, alpha, color, group, linetype, size

j <- geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps

k <- data.frame(murder = USArrests\$Murder, state = tolower(row.names(USArrests)))

map <- map_data("state")
k <- ggplot(data, aes(fill = murder))

k <- geom_map(aes(map_id = state), map = map) + expand_limits(x = map\$long, y = map\$lat)

map_id, alpha, color, fill, linetype, size

l <- geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE) x, y, alpha, fill

l <- geom_tile(aes(fill = z)) x, y, alpha, color, fill, linetype, size, width

Stats

An alternative way to build a layer.

A stat builds new variables to plot (e.g., count, prop).



Visualize a stat by changing the default stat of a geom function, **geom(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function). Use **name_** syntax to map stat variables to aesthetics.

geom to use stat function geom mappings
i <- stat_density_2d(aes(fill = ..level..), geom = "polygon") variable created by stat

c <- stat_bin(binwidth = 1, boundary = 10)

x, y, ..count.., ..density.., ..ndensity..

c <- stat_count(width = 1) x, y, ..count.., ..prop..

x, y, ..count.., ..density.., ..prop..

e <- stat_bin_2d(bins = 30, drop = T)

x, y, fill, ..count.., ..density..

e <- stat_bin_hex(bins = 30) x, y, fill, ..count.., ..density..

e <- stat_density_2d(contour = TRUE, n = 100)

x, y, color, size, ..level..

e <- stat_ellipse(level = 0.95, segments = 51, type = "t")

l <- stat_contour(aes(z = z), x, y, z, order = ..level..)

l <- stat_summary_hex(aes(z = z), bins = 30, fun = mean)

x, y, z, fill, ..value..

l <- stat_summary_2d(aes(z = z), bins = 30, fun = mean)

x, y, z, fill, ..value..

f <- stat_boxplot(coef = 1.5)

x, y, ..lower.., ..middle.., ..upper.., ..width.., ..ymin.., ..ymax..

f <- stat_ydensity(kernel = "gaussian", scale = "area") x, y, ..density.., ..scaled.., ..area.., ..violinwidth.., ..width..

e <- stat_ecdf(n = 40) x, y, ..x.., ..y..

e <- stat_quantile(quantiles = c(0.1, 0.9), formula = y ~ log(x), method = "qt") x, y, ..quantile..

e <- stat_smooth(method = "lm", formula = y ~ x, se = T, level = 0.95) x, y, ..se.., ..x.., ..ymin.., ..ymax..

ggplot() <- xlim(5, 5) + stat_function(fun = dnorm, n = 20, geom = "point") x | ..x.., ..y..

ggplot() <- stat_qq(aes(sample = 1:100)) x, y, sample, ..sample.., ..theoretical..

e <- stat_sum(x, y, size, ..prop..)

e <- stat_summary(fun.data = "mean_cl_boot")

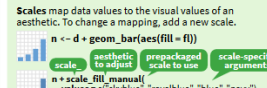
h <- stat_summary_bin(fun = "mean", geom = "bar")

e <- stat_identity()

e <- stat_unique()

Scales

Override defaults with scales package.



Visualize a stat by changing the default stat of a geom function, **geom_bar(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function). Use **name_** syntax to map stat variables to aesthetics.

geom to use stat function geom mappings
i <- stat_density_2d(aes(fill = ..level..), geom = "polygon") variable created by stat

c <- stat_bin(binwidth = 1, boundary = 10)

x, y, ..count.., ..density.., ..ndensity..

c <- stat_count(width = 1) x, y, ..count.., ..prop..

x, y, ..count.., ..density.., ..prop..

e <- stat_bin_2d(bins = 30, drop = T)

x, y, fill, ..count.., ..density..

e <- stat_bin_hex(bins = 30) x, y, fill, ..count.., ..density..

e <- stat_density_2d(contour = TRUE, n = 100)

x, y, color, size, ..level..

e <- stat_ellipse(level = 0.95, segments = 51, type = "t")

l <- stat_contour(aes(z = z), x, y, z, order = ..level..)

l <- stat_summary_hex(aes(z = z), bins = 30, fun = mean)

x, y, z, fill, ..value..

l <- stat_summary_2d(aes(z = z), bins = 30, fun = mean)

x, y, z, fill, ..value..

f <- stat_boxplot(coef = 1.5)

x, y, ..lower.., ..middle.., ..upper.., ..width.., ..ymin.., ..ymax..

f <- stat_ydensity(kernel = "gaussian", scale = "area") x, y, ..density.., ..scaled.., ..area.., ..violinwidth.., ..width..

e <- stat_ecdf(n = 40) x, y, ..x.., ..y..

e <- stat_quantile(quantiles = c(0.1, 0.9), formula = y ~ log(x), method = "qt") x, y, ..quantile..

e <- stat_smooth(method = "lm", formula = y ~ x, se = T, level = 0.95) x, y, ..se.., ..x.., ..ymin.., ..ymax..

ggplot() <- xlim(5, 5) + stat_function(fun = dnorm, n = 20, geom = "point") x | ..x.., ..y..

ggplot() <- stat_qq(aes(sample = 1:100)) x, y, sample, ..sample.., ..theoretical..

e <- stat_sum(x, y, size, ..prop..)

e <- stat_summary(fun.data = "mean_cl_boot")

h <- stat_summary_bin(fun = "mean", geom = "bar")

e <- stat_identity()

e <- stat_unique()

Coordinate Systems

r <- d + geom_bar() + coord_cartesian(xlim = c(0, 5), ylim = 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.

ggplot(mpg, aes(y = fl)) + geom_bar() Flip cartesian coordinates by switching x and y aesthetic mappings.

r <- coord_polar(theta = "x", direction = 1) theta, start, direction - Polar coordinates.

r <- coord_trans(y = "sqrt") - x, y, xlim, ylim Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

pi <- coord_quickmap() pi <- coord_map(projection = "ortho", orientation = c(41, -74, 0)) - projection, xlim, ylim Map projections from the maptools package (mercator (default), aequalarea, 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.

e <- geom_point(position = "jitter") Add random noise to X and Y position of each element to avoid overplotting.

e <- scale_label(position = "nudge") Nudge labels away from points.

s <- geom_bar(position = "stack") Stack elements on top of one another.

Each position adjustment can be recast as a function with manual width and height arguments: s <- geom_bar(position = position_dodge(width = 1))

Themes

r <- theme_bw() White background with grid lines.

r <- theme_classic() Minimal theme.

r <- theme_gray() Grey background (default theme).

r <- theme_minimal() Minimal theme.

r <- theme_dark() Dark for contrast.

r <- theme() Customize aspects of the theme such as axis, legend, panel, and facet properties.

r <- theme(panel.background = element_rect(fill = "blue"))

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(cols = vars(fl)) Facet into columns based on fl.

t <- facet_grid(rows = vars(year)) Facet into rows based on year.

t <- facet_grid(rows = vars(year), cols = vars(fl)) Facet into both rows and columns.

t <- facet_wrap(vars(fl)) Wrap facets into a rectangular layout.

Set scales to let axis limits vary across facets.

t <- facet_grid(rows = vars(drv), cols = vars(fl), scales = "free") x and y axis limits adjust to individual facets.

t <- facet_grid(rows = vars(fl), scales = "free_y") x-axis limits adjust "free_y" - y-axis limits adjust

Set labeller to adjust facet labels:

t <- facet_grid(cols = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)

t <- facet_grid(rows = vars(fl), labeller = label_both)



essencemediacom
business science


Jarek Dejneka, Managing Partner
jaroslaw.dejneka@essencemediacom.com

Bartek Kowalski, Business Science Director
bartosz.kowalski@essencemediacom.com

 mbs@mediacom.com

 [@MBSWarsaw](https://www.instagram.com/MBSWarsaw)

 [@MBSWarsaw](https://twitter.com/MBSWarsaw)

 business-science.pl

***„An investment in knowledge
always pays the best interest”***

Benjamin Franklin

