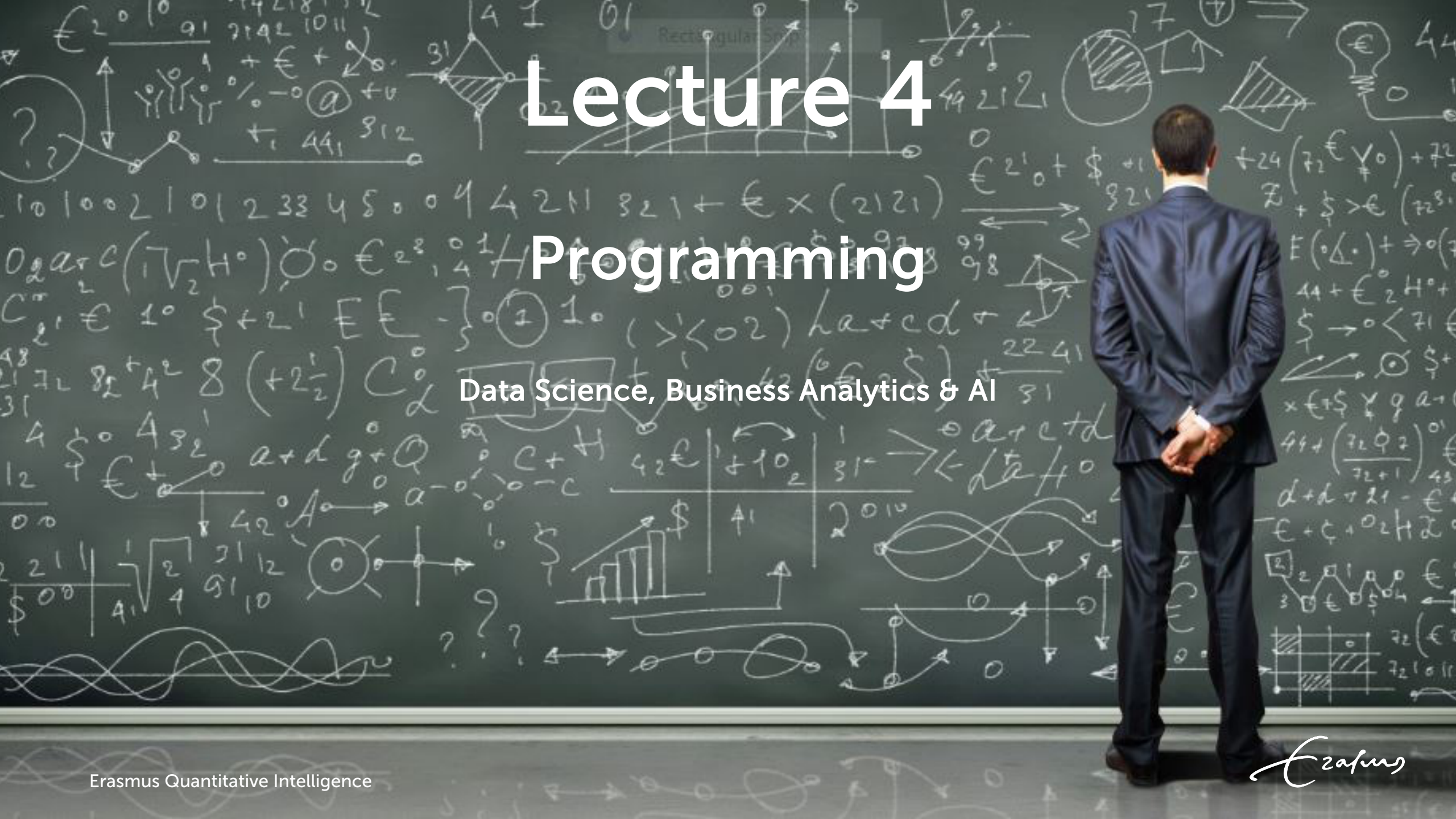


Lecture 4

Programming

Data Science, Business Analytics & AI

Erasmus Quantitative Intelligence

TERUGBLIK VORIGE WEKEN

- Scripts schrijven in R
- Data importeren (csv, Excel, RData)
- Beschrijvende statistieken
- Data visualiseren
- Data transformeren
- Data combineren
- Data ‘netjes’ maken

RECAP FUNCTIES IN R

- Er zijn allerlei ingebouwde functies waarmee je snel informatie uit je data kunt halen. Bijv:

mean()	gemiddelde
sd()	standaarddeviatie
median()	median
min(), max()	minimum en maximum
which.min(), which.max()	de positie (index) van de minimale/maximale waarde
table()	tabel met frequenties voor een categorische variabele
sum()	som
is.na()	TRUE voor een missende waarde, anders FALSE

- Daarnaast veel functies beschikbaar in allerlei packages.

RECAP VISUALISATIE

In R kunnen we eenvoudig grafieken en diagrammen maken met het package 'ggplot2'.

Staafdiagram:

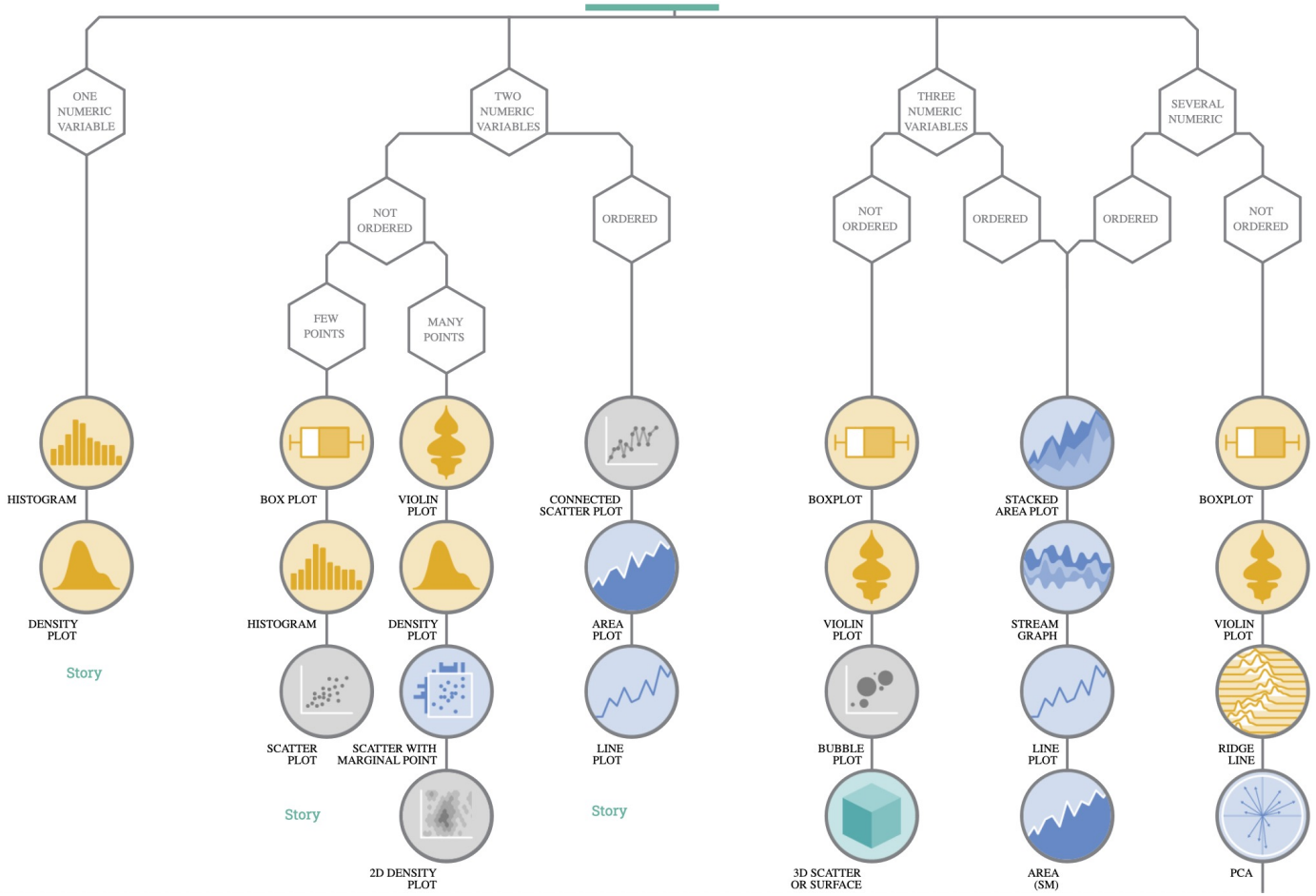
```
> ggplot(data, aes(x=variable,fill=variable))+ geom_bar() + theme_classic()
+ labs(title="your title", x="x axis", y="y axis")
```

Andere plots:

- Lijnplot → geom_line()
- Scatterplot → geom_point()
- Boxplot → geom_boxplot()
- Histogram → geom_histogram()

INSPIRATIE VOOR VERSCHILLENDE SOORTEN PLOTS

Zie bijv. data-to-viz.com of r-graph-gallery.com



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.

required

```
ggplot (data = <DATA>) +
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
 stat = <STAT>, position = <POSITION>) +
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>
```

Not required, sensible defaults supplied

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 = "dotdash", 5 = "longdash", 6 = "twodash")

lineend - 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")

RStudio

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, unemploy))
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, xend, y, yend, alpha, angle, color, curvature, linetype, size

a + geom_path(lineend = "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

b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)**)** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)**)** - x, ymax, ymin, 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(fl))

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 = "bl"**)** 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

e + geom_jitter(height = 2, width = 2**)** x, y, alpha, color, fill, shape, size

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, unemploy))

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(fatten = 2**)** - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width Also **geom_errorbarh()**.

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

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

maps

data <- data.frame(murder = USArrests$Murder, state = tolower(rownames(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

THREE VARIABLES

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

l + geom_contour(aes(z = 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

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

RECAP DATA TRANSFORMEREN

- Eenvoudig met tidyverse en pipes

```
flights_gains <- flights %>%
       filter(month == 3 & day == 13) %>%
       select(carrier, origin, dest, dep_delay, arr_delay) %>%
       mutate(gain = dep_delay - arr_delay) %>%
       group_by(dest) %>%
       summarise(mean_gain = mean(gain, na.rm = TRUE), max_gain =
                     max(gain, na.rm = TRUE),   nr_flights = n())
```

Data Wrangling with dplyr and tidyr

Cheat Sheet

RStudio

Tidy Data - A foundation for wrangling in R

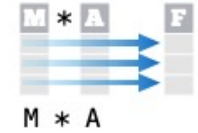
In a tidy data set:

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

&

Each **observation** is saved in its own **row**

Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.



Syntax - Helpful conventions for wrangling

dplyr::**tbl_df(iris)**

Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

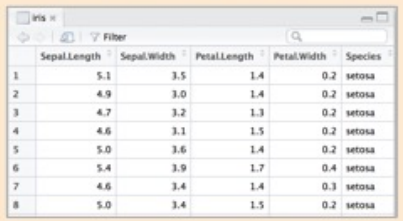
```
Source: local data frame [150 x 5]

   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: Petal.Width (dbl),
  Species (fctr)
```

dplyr::**glimpse(iris)**

Information dense summary of tbl data.

utils::**View(iris)**

View data set in spreadsheet-like display (note capital V).



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa

dplyr::**%>%**

Passes object on left hand side as first argument (or . argument) of function on righthand side.

`x %>% f(y)` *is the same as* `f(x, y)`

`y %>% f(x, ., z)` *is the same as* `f(x, y, z )`

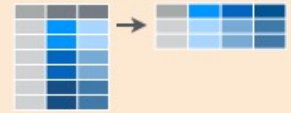
"Piping" with %>% makes code more readable, e.g.

```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

Reshaping Data - Change the layout of a data set

tidyr::**gather(cases, "year", "n", 2:4)**

Gather columns into rows.



tidyr::**spread(pollution, size, amount)**

Spread rows into columns.

tidyr::**separate(storms, date, c("y", "m", "d"))**

Separate one column into several.

tidyr::**unite(data, col, ..., sep)**

Unite several columns into one.

dplyr::**data_frame(a = 1:3, b = 4:6)**

Combine vectors into data frame (optimized).

dplyr::**arrange(mtcars, mpg)**

Order rows by values of a column (low to high).

dplyr::**arrange(mtcars, desc(mpg))**

Order rows by values of a column (high to low).

dplyr::**rename(tb, y = year)**

Rename the columns of a data frame.

Subset Observations (Rows)

dplyr::**filter(iris, Sepal.Length > 7)**

Extract rows that meet logical criteria.

dplyr::**distinct(iris)**

Remove duplicate rows.

dplyr::**sample_frac(iris, 0.5, replace = TRUE)**

Randomly select fraction of rows.

dplyr::**sample_n(iris, 10, replace = TRUE)**

Randomly select n rows.

dplyr::**slice(iris, 10:15)**

Select rows by position.

dplyr::**top_n(storms, 2, date)**

Select and order top n entries (by group if grouped data).

Logic in R - ?Comparison, ?base::Logic

<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&, |, !, xor, any, all	Boolean operators

Subset Variables (Columns)

dplyr::**select(iris, Sepal.Width, Petal.Length, Species)**

Select columns by name or helper function.

Helper functions for select - ?select

select(iris, contains("."))
Select columns whose name contains a character string.

select(iris, ends_with("Length"))
Select columns whose name ends with a character string.

select(iris, everything())
Select every column.

select(iris, matches(".t."))
Select columns whose name matches a regular expression.

select(iris, num_range("x", 1:5))
Select columns named x1, x2, x3, x4, x5.

select(iris, one_of(c("Species", "Genus")))
Select columns whose names are in a group of names.

select(iris, starts_with("Sepal"))
Select columns whose name starts with a character string.

select(iris, Sepal.Length:Petal.Width)
Select all columns between Sepal.Length and Petal.Width (inclusive).

select(iris, -Species)
Select all columns except Species.

devtools::install_github("rstudio/EDAWR") for data sets

Learn more with browseVignettes(package = c("dplyr", "tidyr")) • dplyr 0.4.0• tidyr 0.2.0 • Updated: 1/15

TERUGBLIK OEFENINGEN

Werken met datums

Programming

Data Science, Business Analytics & AI

WERKEN MET DATUMS EN TIJDEN

- Veel datasets bevatten datums of tijden
- De lubridate package bevat nuttige functies om te rekenen met datums en tijden
- Met functies als dmy(), ymd(), ymd_hms() kun je karakters omzetten naar een tijdsobject
  - … mits het format van de karakters overeenkomt met dat van de functie
- Met eenvoudige wiskundige operaties kun je vervolgens tijden optellen of tijdsverschillen berekenen

```
> install.packages("lubridate")
> library(lubridate)
```

DATA TYPE VOOR DATUMS

- Date-times als datatype voor datums

- POSIXct: aantal seconden sinds 1 januari 1970, in combinatie met tijdzone
  - Efficiënt voor opslag en berekeningen
  - Negatieve waarden corresponderen met momenten voor 1 januari 1970

- POSIXlt: jaren, dagen, uren, minuten, etc. worden separaat opgeslagen.
  - Makkelijker leesbaar, maar minder efficiënt qua opslag

- We kunnen waarden parsen naar Date-times met functies als as_datetime() en ymd()

TIJDSPERIODEN

- Difftimes als datatype voor tijdsperioden
- Tijdsperioden kunnen opgeteld worden bij Date-times

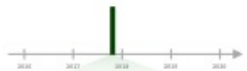
- Period: geeft een duur op basis van 'kloktijd', bijv hours(1)
- Duration: geeft een duur op basis van 'daadwerkelijke tijd', bijv dhours(1) (dit is waar we meestal in geïnteresseerd zijn)

- NB. Periods en durations zijn niet geassocieerd met een specifieke start en eindtijd

- Interval: een tijdsinterval geassocieerd met een specifieke start en eindtijd: as.interval()

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"

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"

12:00:00

An hms is a **time** stored as the number of seconds since 00:00:00

*t <- hms::**as.hms**(85)*
00:01:25

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 — **ydm_hms**(), **ydm_hm**(), **ydm_h**(). *ydm_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**(), **ydm**(). *ymd(20170131)*

July 4th, 2000 — **mdy**(), **myd**(). *mdy("July 4th, 2000")*

4th of July '99 — **dmy**(), **dym**(). *dmy("4th of July '99")*

2001: Q3 — **yq**() Q for quarter. *yq("2001: Q3")*

2:01 — hms::**hms**() Also lubridate::**hms**(), **hm**() and **ms**(), which return periods.* *hms::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 tz (defaults to system tz). *now()*

today(tzone = "") Current date in a tz (defaults to system tz). *today()*

fast_strptime() Faster strptime. *fast_strptime('9/1/01', '%y/%m/%d')*

parse_date_time() Easier strptime. *parse_date_time("9/1/01", "ymd")*

GET AND SET COMPONENTS

Use an accessor function to get a component.

d ## "2017-11-28"
day(d) ## 28

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

day(d) <- 1
d ## "2017-11-01"

2018-01-31 11:59:59 — **date**(x) Date component. *date(dt)*

2018-01-31 11:59:59 — **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)*

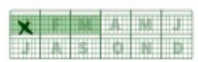
2018-01-31 11:59:59 — **day**(x) Day of month. *day(dt)*
wday(x,label,abbr) Day of week.
qday(x) Day of quarter.

2018-01-31 11:59:59 — **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)*

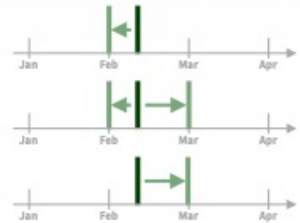
am(x) Is it in the am? *am(dt)*
pm(x) Is it in the pm? *pm(dt)*

dst(x) Is it daylight savings? *dst(d)*

leap_year(x) Is it a leap year? *leap_year(d)*

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
   sf(ymd("2010-04-05"))
   ## [1] "Created Monday, Apr 05, 2010 00:00"

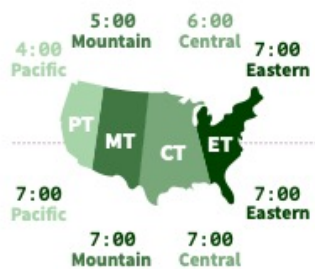
Tip: use a date with day > 12

Time Zones

R recognizes ~600 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()*



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")*

OEFENING

- Maak een nieuwe kolom “arr_delay” met daarin de tijden in minuten tussen de “scheduled arrival time” en daadwerklijke “arrival time”

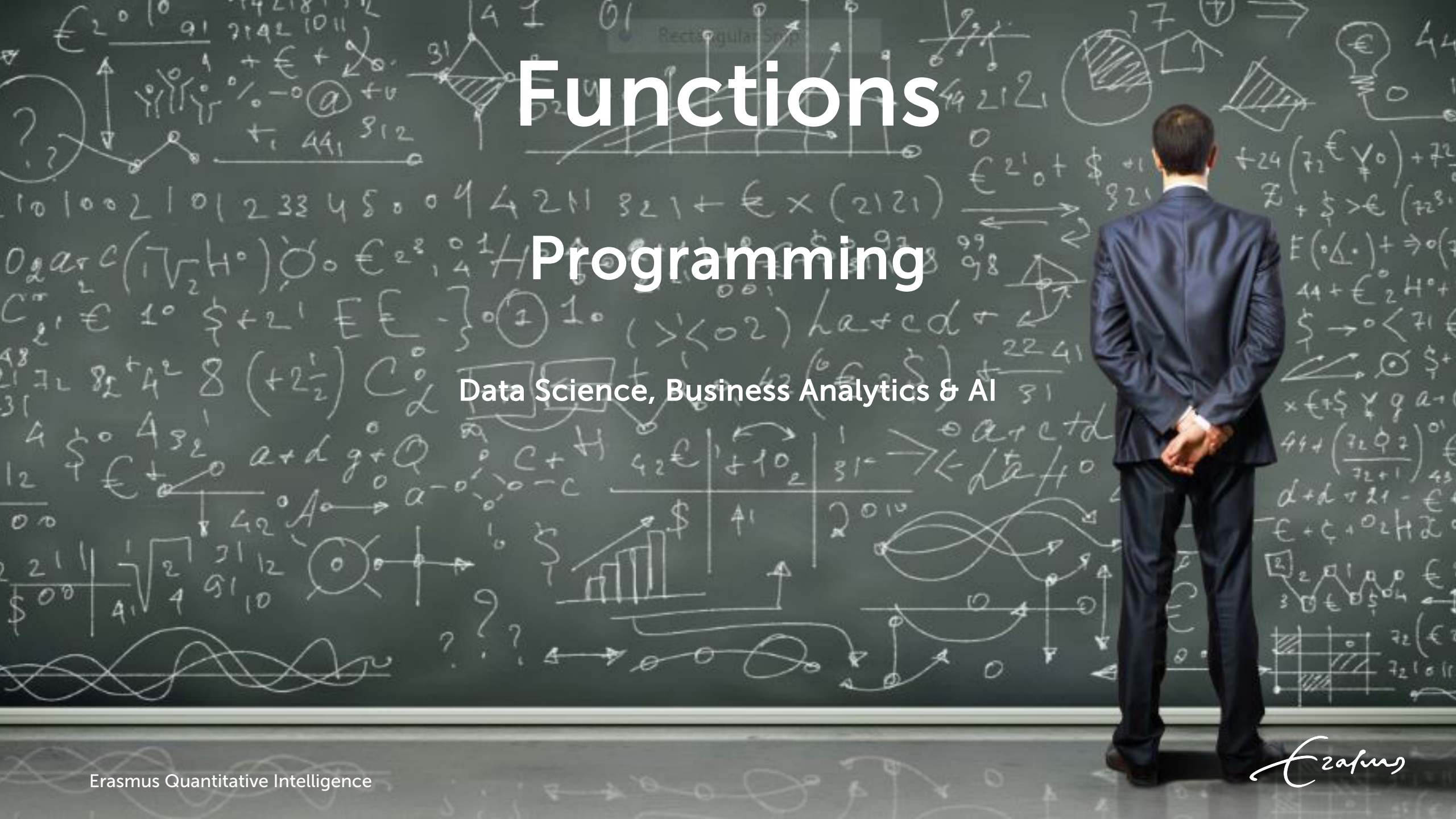
- Stel: we willen weten wat de gemiddelde vertraging (“arr_delay”) is voor vluchten die vertrekken vanaf JFK. Hoe doe je dit?

- Wat als we voor **ieder vertrekvliegveld** de gemiddelde vertraging willen weten?

Functions

Programming

Data Science, Business Analytics & AI



FUNCTIES IN R

- Tot nu toe hebben we telkens bestaande functies gebruikt om een actie uit te voeren:
  - c(5,2,3,1)
  - rbind(1:5, 2:10)
  - getwd()
  - str(BenAndJerry)
  - is.na(BenAndJerry$coupon_value)
  - wday(flights$arr_time)
  - flights %>% mutate(dep_delay = dep_time – sched_dep_time)
  - mean(flights$dep_delay, na.rm=TRUE)
- Elke functie heeft een naam, gevolgd door ronde haken met (vaak) 1 of meer inputs
- De functie voert acties uit met de inputs en geeft je uiteindelijk 1 of meer outputs terug

EEN EIGEN FUNCTIE SCHRIJVEN

- Je kunt ook een eigen functie schrijven, bijvoorbeeld voor een reeks operaties die je vaak wilt herhalen voor verschillende data
- Een eigen functie bestaat uit:
  - Een naam, gevolgd door "<- function(…)"
  - Namen voor de inputvariabelen van je functie, tussen de ronde haken
  - Een function body tussen accolades, met daarin alle instructies om de inputs te bewerken tot een gewenste output
  - (Meestal) een return statement aan het eind van de function body, waarin de output "teruggegeven" wordt aan de gebruiker

Ken je functies die geen data object teruggeven?

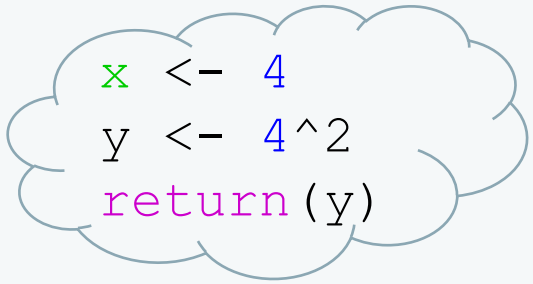
```
square <- function(x) {
  y <- x^2
  return(y)
}
```

EEN EIGEN FUNCTIE AANROEPEN

- Zodra je je zelfgeschreven functie runt in de console, gebeurt er nog niet direct (zichtbaar) iets, maar…
- Vanaf nu kun je je functie wel gebruiken net als iedere andere functie, met naam(inputs)
- Als inputs gebruik je nu daadwerkelijke waarden
- Op de achtergrond doet R nu het volgende:
  1. De inputvariabelen in je functiebeschrijving worden vervangen door de inputwaarden die je nu hebt meegegeven
  2. De instructies in je function body worden uitgevoerd met deze inputwaarden
  3. De berekende output wordt teruggegeven; deze wordt zichtbaar in de console

```
square <- function(x) {
  y <- x^2
  return(y)
}
```

```
> square(4)
```




```
[1] 16
```

OEFENING

Schrijf een eigen functie met de naam ‘mijn_functie’. De functie moet als input een waarde ‘a’ hebben en als output de wortel van ‘a+10’.

Schrijf daarna een functie met de naam ‘mijn_functie2’. De functie moet als input een ggplot2 object ‘p’ hebben en aan die plot de theme_economist toevoegen. Vervolgens wordt de plot teruggegeven als output.

LATEN WE NOG EENS KIJKEN NAAR DATATRANSFORMATIES

```
flights %>%
       mutate(arr_time       = ymd_hm(arr_time),
              sched_arr_time = ymd_hm(sched_arr_time),
              arr_delay      = arr_time - sched_arr_time) %>%
       filter(!is.na(arr_delay), origin == "JFK") %>%
       summarize(av_arrival_delay = mean(arr_delay)/dminutes(1))
```

LATEN WE NOG EENS KIJKEN NAAR DATATRANSFORMATIES

```
flights %>%
      mutate(arr_time       = ymd_hm(arr_time),
             sched_arr_time = ymd_hm(sched_arr_time),
             arr_delay      = arr_time - sched_arr_time,
             dep_time       = ymd_hm(dep_time),
             sched_dep_time = ymd_hm(sched_dep_time),
             dep_delay      = dep_time - sched_dep_time) %>%
      filter(!is.na(arr_delay), origin == "JFK") %>%
      summarize(av_arrival_delay = mean(arr_delay)/dminutes(1),
                av_dep_delay     = mean(dep_delay)/dminutes(1),
                n_flights        = n())
```

LATEN WE NOG EENS KIJKEN NAAR DATATRANSFORMATIES

```
flights %>%
      mutate(arr_time       = ymd_hm(arr_time),
             sched_arr_time = ymd_hm(sched_arr_time),
             arr_delay      = arr_time - sched_arr_time,
             dep_time       = ymd_hm(dep_time),
             sched_dep_time = ymd_hm(sched_dep_time),
             dep_delay      = dep_time - sched_dep_time) %>%
      filter(!is.na(arr_delay)) %>%
      group_by(dest) %>%
      summarize(av_arrival_delay = mean(arr_delay)/dminutes(1),
                av_dep_delay     = mean(dep_delay)/dminutes(1),
                n_flights        = n())
```

LATEN WE NOG EENS KIJKEN NAAR DATATRANSFORMATIES

```
flights %>%
      mutate(arr_time       = ymd_hm(arr_time),
             sched_arr_time = ymd_hm(sched_arr_time),
             arr_delay      = arr_time - sched_arr_time,
             dep_time       = ymd_hm(dep_time),
             sched_dep_time = ymd_hm(sched_dep_time),
             dep_delay      = dep_time - sched_dep_time) %>%
      filter(!is.na(arr_delay)) %>%
      group_by(origin, dest) %>%
      summarize(av_arrival_delay = mean(arr_delay)/dminutes(1),
                av_dep_delay     = mean(dep_delay)/dminutes(1),
                n_flights        = n())
```

LATEN WE NOG EENS KIJKEN NAAR DATATRANSFORMATIES

```
flights %>%
      mutate(arr_time       = ymd_hm(arr_time),
             sched_arr_time = ymd_hm(sched_arr_time),
             arr_delay      = arr_time - sched_arr_time,
             dep_time       = ymd_hm(dep_time),
             sched_dep_time = ymd_hm(sched_dep_time),
             dep_delay      = dep_time - sched_dep_time) %>%
      filter(!is.na(arr_delay)) %>%
      group_by(dest) %>%
      summarize(av_arrival_delay = mean(arr_delay)/dminutes(1),
                av_dep_delay     = mean(dep_delay)/dminutes(1),
                n_flights        = n()) %>%
      arrange(desc(av_arrival_delay))
```

LATEN WE NOG EENS KIJKEN NAA S

Het eerste deel van de bewerkingen is bijna overal hetzelfde

```
flights %>%
      mutate(arr_time       = ymd_hm(arr_time),
             sched_arr_time = ymd_hm(sched_arr_time),
             arr_delay      = arr_time - sched_arr_time,
             dep_time       = ymd_hm(dep_time),
             sched_dep_time = ymd_hm(sched_dep_time),
             dep_delay      = dep_time - sched_dep_time) %>%
      filter(!is.na(arr_delay)) %>%
      group_by(dest) %>%
      summarize(av_arrival_delay = mean(arr_delay)/dminutes(1),
                av_dep_delay     = mean(dep_delay)/dminutes(1),
                n_flights        = n()) %>%
      arrange(desc(av_arrival_delay))
```

LATEN WE NOG EENS KIJKEN NAAR DE STAPPEN

We zouden dit kunnen vervangen door een eigen functie!

```
flights %>%
       prepare_flight_data() %>%
       group_by(dest) %>%
       summarize(av_arrival_delay = mean(arr_delay)/dminutes(1),
                 av_dep_delay     = mean(dep_delay)/dminutes(1),
                 n_flights        = n()) %>%
       arrange(desc(av_arrival_delay))
```

Voordeel: de code wordt overzichtelijker en soortgelijke preparaties kosten voortaan minder typwerk

FUNCTIE VOOR DATAPREPARATIE

```
prepare_flight_data <- function(flight_data) {
  prepared <- flight_data %>%
    mutate(arr_time = ymd_hm(arr_time),
           sched_arr_time = ymd_hm(sched_arr_time),
           dep_time = ymd_hm(dep_time),
           sched_dep_time = ymd_hm(sched_dep_time),
           arr_delay = arr_time - sched_arr_time,
           dep_delay = dep_time - sched_dep_time) %>%
    filter(!is.na(arr_time))
  return(prepared)
}
```

In het script gebruiken we `parse_date_time(arr_time, "%Y-%m-%d %H:%M")`, omdat `ymd_hm` niet altijd werkt

Hier definiëren we onze functie

```
prepared_flights <- prepare_flight_data(flights)
```

Hier gebruiken we onze functie

FUNCTIES MET MEERDERE INPUTS

- Functies kunnen ook meerdere inputs hebben, die samen nodig zijn om de output te berekenen
- Stel: we willen de prestaties van een vluchtmaatschappij samenvatten. Voor een specifieke maatschappij, bijv. United Airlines ("UA") konden we dit als volgt doen:

```
flights %>%
   prepare_flight_data() %>%
   filter(carrier == "UA") %>%
   summarize(av_arr_delay = mean(arr_delay),
             av_dep_delay = mean(dep_delay),
             n_flights = n())
```

- Als we eenzelfde berekening willen herhalen voor andere maatschappijen, is het nuttig om een functie te maken. Als inputs hebben we dan nodig:
  - Een dataframe met vluchtdata
  - De naam van de vluchtmaatschappij

FUNCTIES MET MEERDERE INPUTS

Hier definiëren we onze functie, genaamd “summarize_carrier”

```
summarize_carrier <- function(flight_data, car) {
   carrier_summary <- flight_data %>%
      prepare_flight_data() %>%
      filter(carrier == car) %>%
      summarize(av_arr_delay = mean(arr_delay),
                av_dep_delay = mean(dep_delay),
                n_flights = n())
   return(carrier_summary)
}
```

“UA” is nu vervangen door de algemene inputvariabele

```
> summarize_carrier(flights, “UA”)
# A tibble: 1 x 3
  av_arr_delay   av_dep_delay   n_flights
  <drtn>         <drtn>             <int>
1 236.748 secs   722.0378 secs      57916
```

Hier roepen we de functie aan

OEFENING

Voeg een derde input toe aan de vorige functie, “min_dist”, die alleen de vluchten filtert met een minimale afstand van “min_dist”.

De output is nog steeds een samenvatting van een specifieke vluchtmaatschappij, maar alleen voor de gefilterde subset van vluchten.

FUNCTIES MET MEERDERE OUTPUTS

- Functies kunnen ook meerde outputs hebben, samengebracht in een list
- Wat gebeurt er in de functie hiernaast?

```
summarize_carrier <- function(flight_data, car) {
  carrier_summary <- flight_data %>%
    prepare_flight_data() %>%
    filter(carrier == car)

  carrier_summary_short <- carrier_summary %>%
    filter(distance <= 3000) %>%
    summarize(av_arr_delay = mean(arr_delay),
              av_dep_delay = mean(dep_delay),
              n_flights = n())

  carrier_summary_long <- carrier_summary %>%
    filter(distance > 3000) %>%
    summarize(av_arr_delay = mean(arr_delay),
              av_dep_delay = mean(dep_delay),
              n_flights = n())

  return(list(short_dist_flights = carrier_summary_short,
              long_dist_flights = carrier_summary_long))
}
```

OEFENING

Als we het aantal vluchten tussen JFK airport en Miami (MIA) willen weten, kunnen we dit als volgt berekenen:

```
flights %>%
  prepare_flight_data() %>%
  filter(origin == "JFK", dest == "MIA") %>%
  summarize(n_flights = n())
```

Schrijf nu een algemene functie genaamd "count_flights" die:

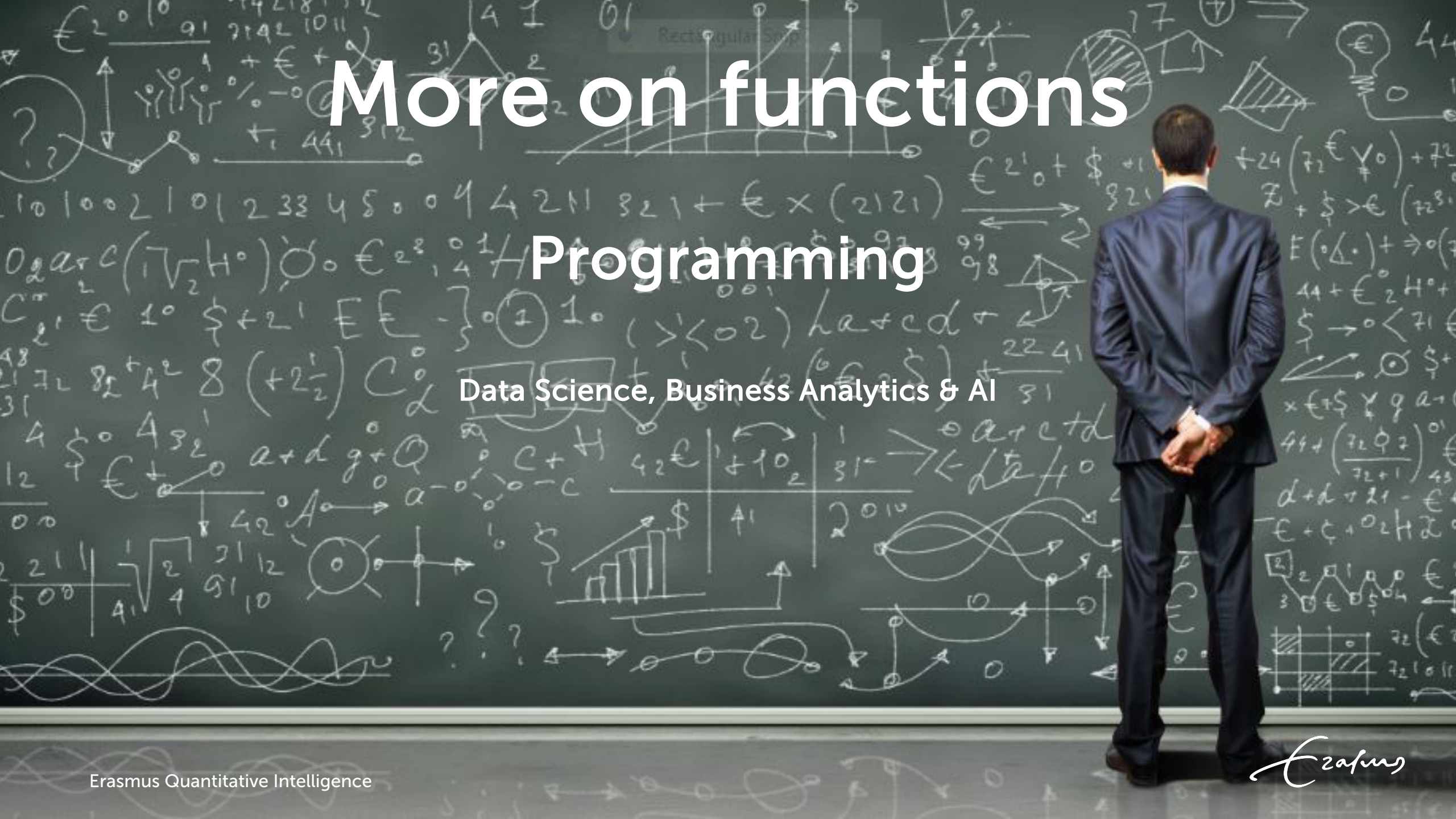
- als inputs (1) een vluchtdataset, (2) een vertekvliegveld en (3) een aankomstvliegveld neemt
- als output aantal vluchten tussen dit paar vliegvelden teruggeeft

Bonusopdracht: geef als extra output de proportie vluchten tussen dit paar vliegvelden met een vertraging van meer dan 30 minuten

More on functions

Programming

Data Science, Business Analytics & AI



What are functions?

All actions are performed by functions

- `c()`, `mean()`, `` `<-` ``
- Takes an object (or objects) and returns a transformed result

Functions are pieces of code dat perform specific actions

- For example: mean() calculates the mean

Functions accept objects as arguments

- Data to compute on and other options that control output

Functions are separate pieces of code with a name

Readability of code

Give consistent names to functions!

Reusability of code

Use same code without copying

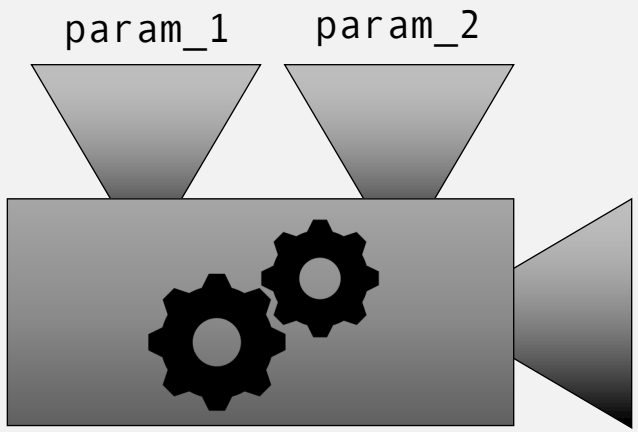
For using functions, it is only needed to know: **"What goes in, what comes out?"**

It is not necessary to exactly know what the function does (it of course is if you write functions yourself)

What are functions? (2)

- every function has 0 or more input values: arguments
- every function has only 1 output value: return value
  can also be `NULL` if no output is needed
- more than 1 output can be returned in a list()-object

- Every function does one thing
- Write separate functions for separate tasks
- A function should be as generic as possible for reusability




```
my_function <-
    function(param_1, param_2){…}
```

Function input

Every input argument has its own name

- `my_function <- function(param_1, param_2) {...}`

When you call a function, the input arguments should be given a value

- `my_function(param_1 = 1, param_2 = 2)`
- `my_function(1, 2)`

Except if a default value is given

- `my_function <- function(param_1, param_2 = 0) {...}`

then:

- `my_function(param_1 = 1)` works
- `my_function(param_1 = 1, param_2 = 2)` overwrites the default value of param_2
- `my_function(param_2 = 2)` Error: argument “param_1” is missing, with no default

Function output

A function always has 1 output: the return value

- If no output is needed, output can be set to `NULL` or equal to the input
  - Example: a function creates a visualization, stored to a .png-file

If you want to return more than 1 output, you store them in a `list()`

- ```
  my_function <- function(param_1, param_2 = 0) {
        som <- param_1 + param_2
        product <- param_1 * param_2
        list(output_1 = som, output_2 = product)
  ```
- ```
  my_function(1, 2)
  ```

Advice:

- Always return the same data type from a function; **be consistent**
- Use informative names on your functions;
  if it is difficult to think of one that represents what your function does, **your function is too complex**
- Make sure your function does **1 thing!**

Function body

- A function consists of a header and a body
  - The header defines the function name and arguments
    - Arguments can have default inputs. If you do not give an input for this argument, the function will use the default.

      Functions with an argument without input or default will crash!
  - The body contains the code needed to get to the desired result
    - The body is between { and }

- Functions have their own function environment
  - Variables in a function cannot be used outside the function
  - When the function is done executing, the functional environment is dropped
  - All you want to keep from a function, should be in output!

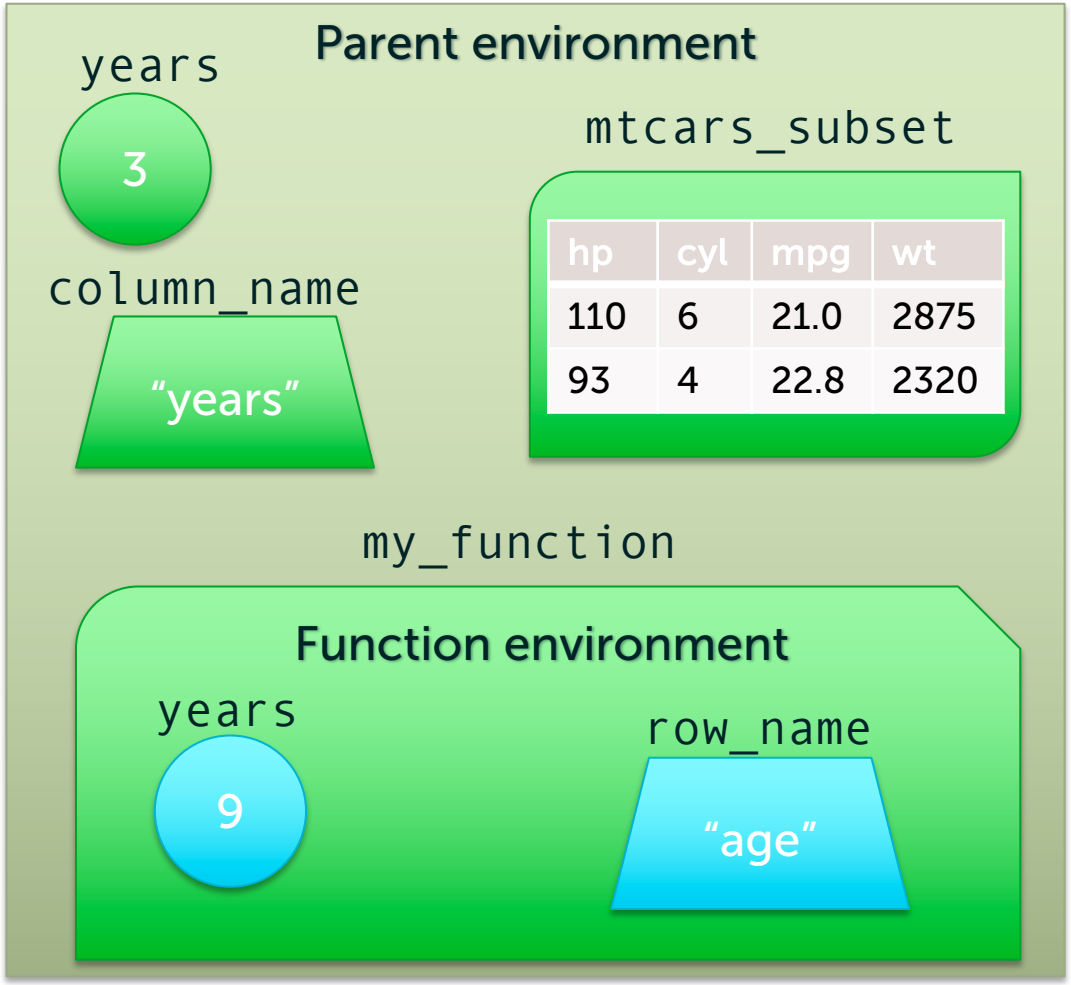
Function body (2)

```
years <- 3
column_name <- "years"

my_function <- function(row_name = "age"){
  years <- 9
  tibble(type = row_name,
         age = years)
}
```

This does not change `years` in the parent environment!

function body between { and }

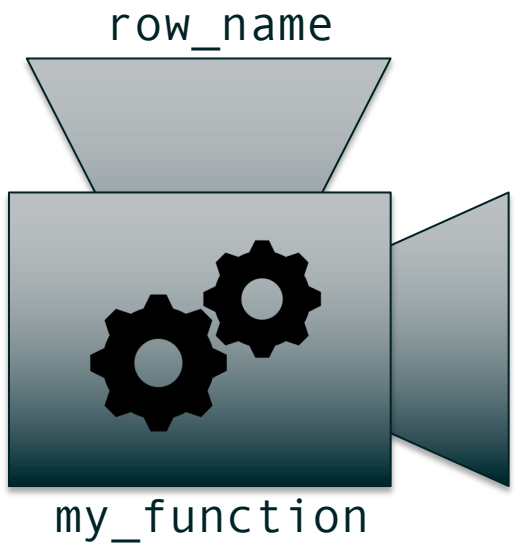
The result of the last line is automatically the return value!



Function body (3)

```
years <- 3
column_name <- "years"
```

```
my_function2 <- function(row_name = "age"){
    years <- 9
    tibble(type = row_name,
           age = years)
}
```




```
my_function2()
```

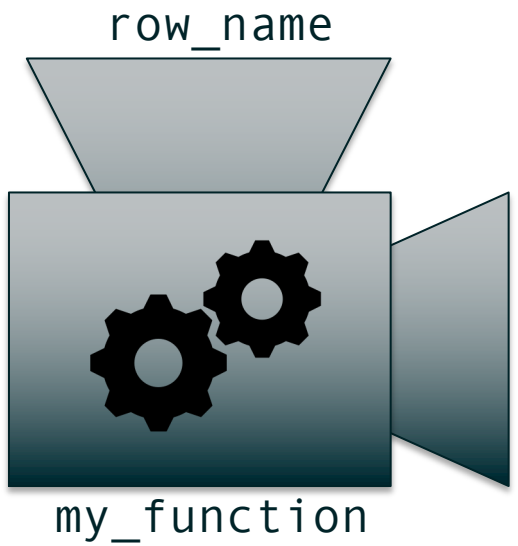
Function call and execution

Since we do not give a value for `row_name` the default is used, namely "`age`".

Function body (4)

```
years <- 3
column_name <- "years"
```

```
my_function2 <- function(row_name = ”age"){
    years <- 9
    tibble(type = row_name,
           age = years)
}
```




```
my_function2(“bla”)
```

Function call and execution

Now, we overwrite the default with ”bla”

Function body (5)

- You can use a variable from the parent environment inside the function
  - For clarity, it is suggested always to use function arguments, such that the function does not depend on the parent environment
- You can override a variable from the parent enivronment inside a function, but this will not change the value *outside* the function
- You cannot use variables which are local to the function in the parent environment

Functions

Functions in isolation

- Using the same arguments will always lead to the same return value
- not dependent on parent environment
- It doesn't matter *where* you call the function
- You can write the function wherever you want, for example in a separate script containing all your functions for this project

You can use arguments from the parent environment, also when not given to the function through input arguments

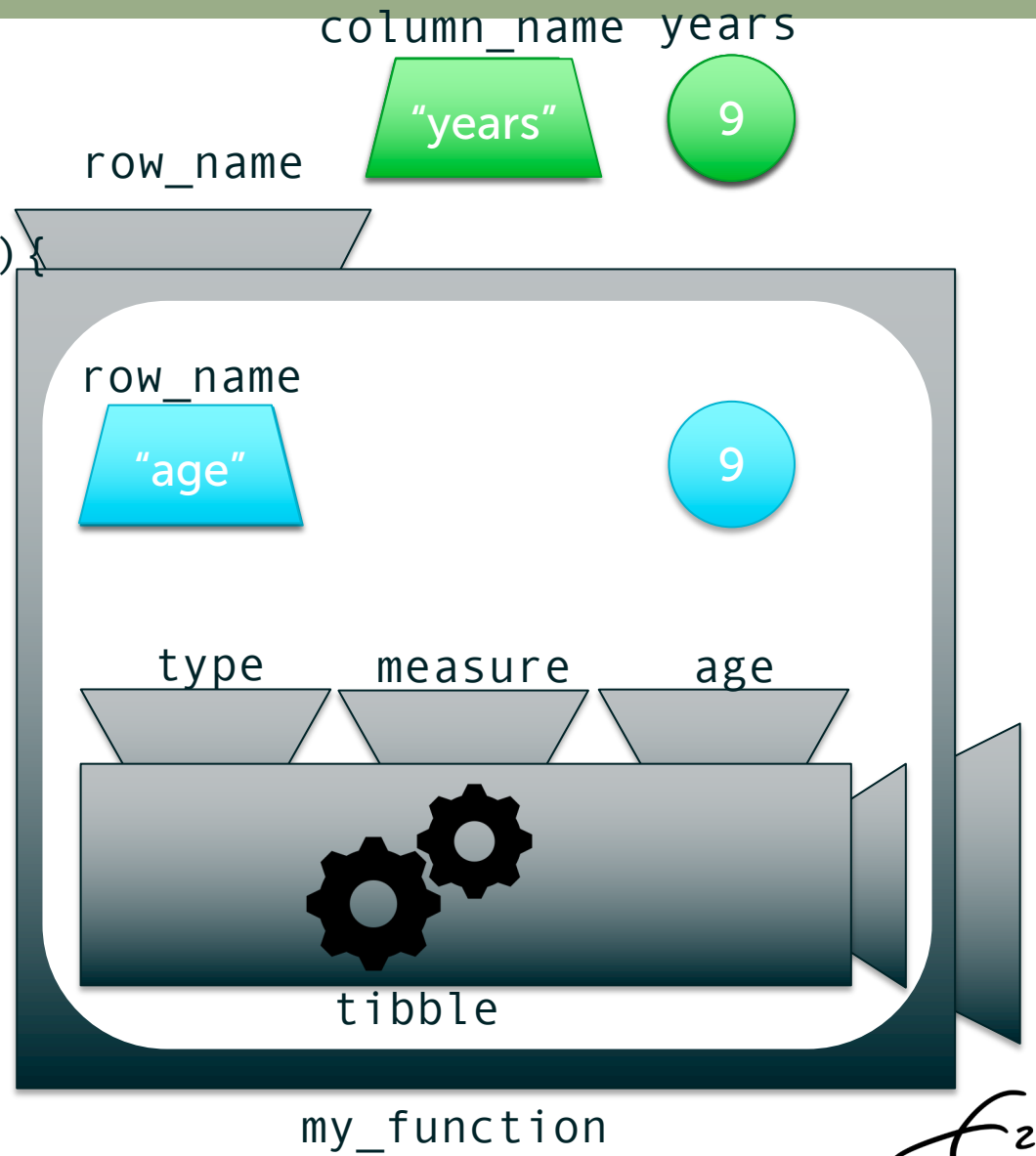
- More difficult to understand the function
- Outcome of the function depends on the current working environment

You can overwrite the parameter in de parent environment using `<<-`

- **This is almost always a bad idea;** upredictable code

Functions (2)

```
years <- 3
column_name <- "years"

my_function3 <- function(row_name = "age"){
   years <<- 9
   tibble(type = row_name,
          measure = column_name,
          age = years)
}

my_function3()
```

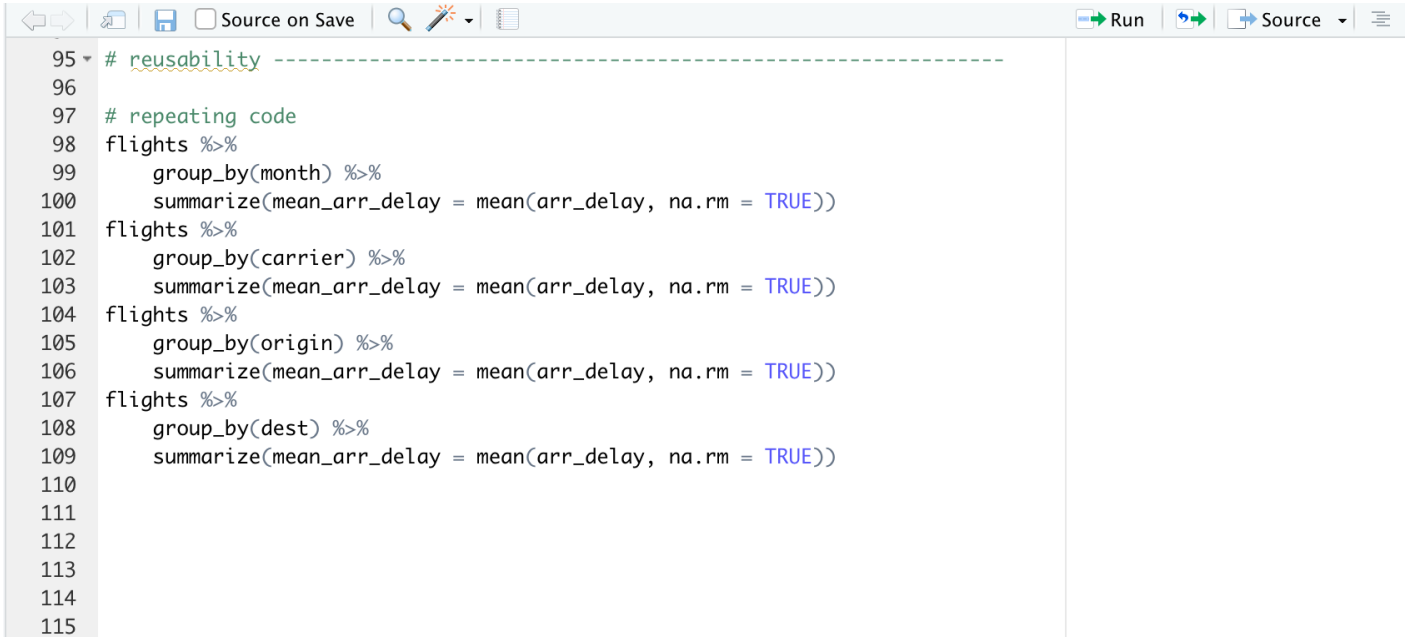


Exercises

- Do Exercise 1 and 2

Reusability

- Copying code is always a bad idea
- Make a function which can repeatedly be used
  - Only one version of code
  - Good for maintenance
  - Good for readability



Reusability (2)

Step 1: Copy one line of code of which you want to create a function

```
flights %>% group_by(month) %>% summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
```

Step 2: Make it a function by adding a function header

```
get_delay_by <- function( ) {
        flights %>% group_by(month) %>% summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
}
```

Step 3: Decide which variables should be arguments to the function

```
get_delay_by <- function(group) {
     flights %>% group_by({{group}}) %>% summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
}
```

Step 4: Replace repeating code by functions

```
delay_by_month <- get_delay_by(month)
delay_by_carrier <- get_delay_by(carrier)
delay_by_origin <- get_delay_by(origin)
delay_by_dest <- get_delay_by(dest)
```

Reusability (3)

Imagine you also want to know the median of arr_delay by a grouping variable.

```
... %>% summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE),
                  median_arr_delay = median(arr_delay, na.rm = TRUE))
```

Do you want to update the code left or right?

Source on Save

```r
# reusability -------------------------------------------------------------

# repeating code
flights %>%
    group_by(month) %>%
    summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
flights %>%
    group_by(carrier) %>%
    summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
flights %>%
    group_by(origin) %>%
    summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
flights %>%
    group_by(dest) %>%
    summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
```

Source on Save

```r
# reusable function
get_delay_by <- function(group) {
    flights %>%
        group_by({{group}}) %>%
        summarise(mean_arr_delay = mean(arr_delay, na.rm = TRUE))
}

delay_by_month <- get_delay_by(month)
delay_by_carrier <- get_delay_by(carrier)
delay_by_origin <- get_delay_by(origin)
delay_by_dest <- get_delay_by(dest)
```

Exercises

- Do Exercise 3 and 4

Summary

- Overcome repeating code by using function

- A function only has one function/role
  - be compact and to-the-point
  - TIP: if it is difficult to choose a name for your function which really represents your function, then your function is probably too large

- Isolate functions as much as possible
  - all input comes via arguments, all output via return values

- Be consistent on output
  - output is always in the same data format

- Be clear on documentation
  - use comments (#) to describe what your function does, what goes in and what comes out
  - TIP: use a Roxygen skeleton (`alt-shift-ctrl-R`)