An Introduction to the R Programming Language

ECP 8506 - Fall 2024

Samuel Callisto, PhD
Senior Scientist, Metrum Research Group
Adjunct Assistant Professor, ECP UMN

METRUM
RESEARCH GROUP

Outline

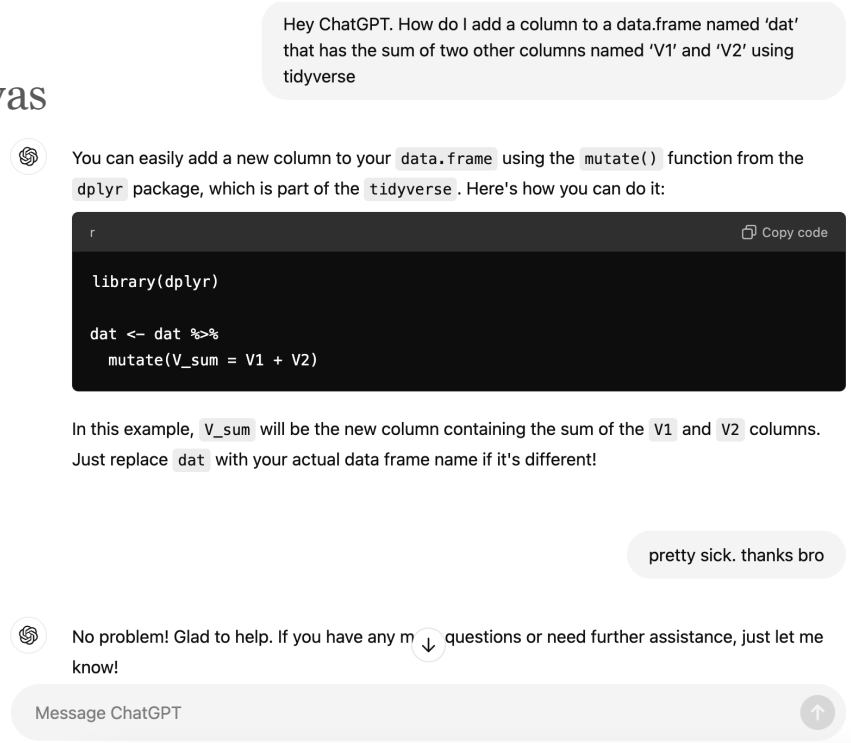
- dplyr/tidyr
  - Manipulating data.frames/tibbles
  - Joins/pivots
- ggplot
  - Dot plots
  - Line plots
  - Box plots
  - Aesthetics
- Functions
- Loops
  - For-loops
  - Map functions

Why is R so hard to learn?

- First and foremost, R is a programming language
  - This is both its greatest strength and largest impediment for new learners
- R is fundamentally different from other statistics tools such as SAS, Stata, SPSS, Excel, etc.
- Many scientists have not taken formal programming courses
- Understanding some basic programming principles will help you become a better R user

If you've never used R. . .

- Watch the youtube lecture posted to Canvas
  - Slides included as backups here
- Google is your friend, ChatGPT is your tutor
- The best way to actually learn any programming language is to do a project that uses it
  - Reproducible data assembly
  - Exploratory data analysis
  - Simple statistical analyses



METRUM RESEARCH GROUP

Tidyverse

- The Tidyverse is a collection of packages specially focused on data science
- The core of the tidyverse collection is made up of the following packages:

tibble	Modern and effective table system
dplyr	Grammar for data manipulation
tidyr	Set of functions to create tidy data
stringr	Function set to work with characters
readr	An easy and fast way to import data
forcats	Tools to easily work with factors

- To install: `install.packages("tidyverse")`
- To load: `library(tidyverse)`

Read and Write Data

base R functions also exist (read.csv), and other data structures are available

- The readr package makes it easy to read or write multiple file formats using functions that start with read_* or write_*

Functions		Description
read_csv() or read_csv2()	write_csv()	comma or semicolon (CSV)
read_delim()	write_delim()	general separator
read_table()	write_table()	whitespace-separated

- The imported tables are of class tibble (tbl_df), a modern version of data.frame from the tibble package

```
> library(readr)
> nmdata <- read_csv("pk-nonmem-example.csv")
> dim(nmdata)
[1] 1488   27
```

argument is a relative path from your working directory

What's a tibble?

- A tibble is a special type of data.frame used by tidyverse
- Includes and displays metadata about each column
- data.frames are automatically converted to tibbles as outputs of tidyverse functions
- For all intents and purposes, tibbles and data.frames can be treated the same
- Can manually convert back and forth using typecasting functions `as_tibble()` or `as.data.frame()`

Pipe %>%

There is also a new pipe operator |> in base R that mostly works the same

- The pipe operator, %>%, allows the user to combine functions without the need to assign the result to a new object
- The pipe operator passes the output of a function to the first argument of the next function
- This way of combining functions allows you to chain together sequential tasks

- Trivial example:

```
1:5 %>% mean()
[1]  3
```

The Treachery of Images, 1929 by Rene Magritte

METRUM RESEARCH GROUP

major data manipulation verbs

Verb	Usage
filter	keep matching row criteria
summarize	calculates summaries, returns reduced output
mutate	add new variables to existing data.frame
select	select columns by name
arrange	reorder rows

```
df <- data.frame(
       ID     = 1:5,
       GENDER = c("MALE", "MALE", "FEMALE",
                  "MALE", "FEMALE"),
       WT     = c(70, 76, 60, 64, 68))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

Verb	Usage
filter	keep matching row criteria
summarize	calculates summaries, returns reduced output
mutate	add new variables to existing data.frame
select	select columns by name
arrange	reorder rows

```
filter(df, GENDER == "FEMALE")
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT
3	FEMALE	60
5	FEMALE	68

common dplyr filter (subset) operators

operator	meaning
==, !=	equal, not equal
>, >=	greater than, greater than or equal to
<, <=	less than, less than or equal to
is.na(), !is.na()	is NA, not NA
!duplicated()	only first value
%in%	in specified values

filter separator	base equivalent	meaning
,	&	and
|	|	or

```
filter(df, ID %in% c(1, 3, 5))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT
1	MALE	70
3	FEMALE	60
5	FEMALE	68

```
filter(df, GENDER == "MALE", WT > 70)
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT
2	MALE	76

```
filter(df, GENDER == "FEMALE" | WT < 70)
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT
3	FEMALE	60
4	MALE	64
5	FEMALE	68

Verb	Usage
filter	keep matching row criteria
summarize	calculates summaries, returns reduced output
mutate	add new variables to existing data.frame
select	select columns by name
arrange	reorder rows

```
df %>%
summarize(meanWT = mean(WT))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

meanWT
67.6

equivalent to. . .

```
summarize(df, meanWT = mean(WT))
```

```
df %>% group_by(GENDER) %>%
  summarize(meanWT = mean(WT)) %>%
  ungroup()
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

GENDER	meanWT
MALE	70
FEMALE	64

You should (almost) always ungroup after grouping, but I will omit it to save space in future slides

```
df %>% group_by(GENDER) %>%
    summarize(meanWT = mean(WT),
              .groups="drop")
```

Groups can also be dropped using summarize arguments

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

GENDER	meanWT
MALE	70
FEMALE	64

```
df %>% group_by(GENDER) %>%
    summarize(meanWT = mean(WT),
    n = n())
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

GENDER	meanWT	n
MALE	70	3
FEMALE	64	2

Verb	Usage
filter	keep matching row criteria
summarize	calculates summaries, returns reduced output
mutate	add new variables to existing data.frame
select	select columns by name
arrange	reorder rows

```
df %>% mutate(meanWT = mean(WT))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

```
df %>%
  group_by(GENDER) %>%
  mutate(meanWT = mean(WT))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT	meanWT
1	MALE	70	70
2	MALE	76	70
3	FEMALE	60	64
4	MALE	64	70
5	FEMALE	68	64

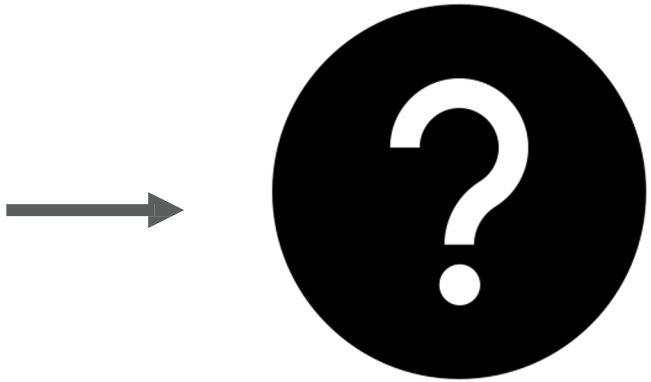
```
df %>% group_by(GENDER) %>%
  mutate(meanWT = mean(WT),
         mWT_LB = meanWT*2.2)
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT	meanWT	mWT_LB
1	MALE	70	70	154
2	MALE	76	70	154
3	FEMALE	60	64	140.8
4	MALE	64	70	154
5	FEMALE	68	64	140.8

```
df %>%
  mutate(ISM = if_else(GENDER == "MALE", 1, 0))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68



A brief aside on conditional statements

- R has several ways to conditionally apply a function or return a value based on some conditional statement
- The most commonly used function in data analysis is if_else()
  - switch statements are implemented using case_when() function (see backups)

conditional statement

returned if FALSE

```
if_else(GENDER == "MALE", 1, 0)
```

function call

returned if TRUE

```
df %>%
  mutate(ISM = if_else(GENDER == "MALE", 1, 0))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT	ISM
1	MALE	70	1
2	MALE	76	1
3	FEMALE	60	0
4	MALE	64	1
5	FEMALE	68	0

Verb	Usage
filter	keep matching row criteria
summarize	calculates summaries, returns reduced output
mutate	add new variables to existing data.frame
select	select columns by name
arrange	reorder rows

```
df2 %>% select(ID, WT)
```

ID	GENDER	WT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

ID	WT
1	70
2	76
3	60
4	64
5	68

```
df2 %>% select(GENDER:meanWT)
```

ID	GENDER	WT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

GENDER	WT	meanWT
MALE	70	67.6
MALE	76	67.6
FEMALE	60	67.6
MALE	64	67.6
FEMALE	68	67.6

```
df2 %>% select(-ID)
```

ID	GENDER	WT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

GENDER	WT	meanWT
MALE	70	67.6
MALE	76	67.6
FEMALE	60	67.6
MALE	64	67.6
FEMALE	68	67.6

```
df2 %>% select(ID, WEIGHT = WT)
```

ID	GENDER	WT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

ID	WEIGHT
1	70
2	76
3	60
4	64
5	68

```
df2 %>% rename(WEIGHT = WT)
```

ID	GENDER	WT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

ID	GENDER	WEIGHT	meanWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

df %>% select(<function>(<arguments>))

df with the following columns:

WEIGHT	WEIGHT_KG	MEAN_WEIGHT	OCC1	OCC2	OCC3	OCC4	HEIGHT

function	meaning	example	columns selected
starts_with	names start with	starts_with("WEIGHT"))	WEIGHT, WEIGHT_KG
ends_with	names ends with	ends_with("GHT")	WEIGHT, MEAN_WEIGHT, HEIGHT
contains	names contains	contains("EI")	WEIGHT, WEIGHT_KG, MEAN_WEIGHT, HEIGHT
matches	regular expression matching	matches("_")	WEIGHT_KG, MEAN_WEIGHT
num_range	specify range of columns with consistent names with numeric suffix	num_range("OCC",1:3)	OCC1, OCC2, OCC3

Verb	Usage
filter	keep matching row criteria
summarize	calculates summaries, returns reduced output
mutate	add new variables to existing data.frame
select	select columns by name
arrange	reorder rows

```
df %>% arrange(WT)
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT
3	FEMALE	60
4	MALE	64
5	FEMALE	68
1	MALE	70
2	MALE	76

lowest weight

highest weight

```
df %>% arrange(desc(WT))
```

ID	GENDER	WT
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	WT
2	MALE	76
1	MALE	70
5	FEMALE	68
4	MALE	64
3	FEMALE	60

highest weight

lowest weight

minor data manipulation verbs

Verb	Usage
distinct	keep all distinct elements per key
slice	return certain rows by number (overall or within group)
rename	rename columns
relocate	move a column within the data frame

```
head(Theoph)
```

Subject	Wt	Dose	Time	conc
1	79.6	4.02	0	0.74
1	79.6	4.02	0.25	2.84
1	79.6	4.02	0.57	6.57
1	79.6	4.02	1.12	10.5
1	79.6	4.02	2.02	9.66
1	79.6	4.02	3.82	8.58

```
Theoph %>%
distinct(Subject)
```

Subject
3
6
7
8
11

```
head(Theoph)
```

Subject	Wt	Dose	Time	conc
1	79.6	4.02	0	0.74
1	79.6	4.02	0.25	2.84
1	79.6	4.02	0.57	6.57
1	79.6	4.02	1.12	10.5
1	79.6	4.02	2.02	9.66
1	79.6	4.02	3.82	8.58

```
Theoph %>%
distinct(Subject,
         .keep_all = TRUE)
```

Subject	Wt	Dose	Time	conc
1	79.6	4.02	0	0.74
2	72.4	4.4	0	0
3	70.5	4.53	0	0
4	72.7	4.4	0	0
5	54.6	5.86	0	0

```
Theoph %>%
group_by(Subject) %>%
slice(1:2)
```

Subject	Wt	Dose	Time	conc
6	80.0	4.00	0.00	0.00
6	80.0	4.00	0.27	1.29
7	64.6	4.95	0.00	0.15
7	64.6	4.95	0.25	0.85
8	70.5	4.53	0.00	0.00
8	70.5	4.53	0.25	3.05
11	65.0	4.92	0.00	0.00
11	65.0	4.92	0.25	4.86
3	70.5	4.53	0.00	0.00
3	70.5	4.53	0.27	4.40

```
Theoph %>%
group_by(Subject) %>%
slice(c(1, n()))
```

Subject	Wt	Dose	Time	conc
6	80.0	4.00	0.00	0.00
6	80.0	4.00	23.85	0.92
7	64.6	4.95	0.00	0.15
7	64.6	4.95	24.22	1.15
8	70.5	4.53	0.00	0.00
8	70.5	4.53	24.12	1.25
11	65.0	4.92	0.00	0.00
11	65.0	4.92	24.08	0.86
3	70.5	4.53	0.00	0.00
3	70.5	4.53	24.17	1.05

```
Theoph %>%
   rename(ID = Subject)
```

ID	Wt	Dose	Time	conc
3	70.5	4.53	0.00	0.00
3	70.5	4.53	0.27	4.40

```
head(Theoph)
```

Subject	Wt	Dose	Time	conc
1	79.6	4.02	0	0.74
1	79.6	4.02	0.25	2.84
1	79.6	4.02	0.57	6.57
1	79.6	4.02	1.12	10.5
1	79.6	4.02	2.02	9.66
1	79.6	4.02	3.82	8.58

```
Theoph %>%
  relocate(Wt, .after=conc)
```

Subject	Dose	Time	conc	Wt
1	4.02	0	0.74	79.6
1	4.02	0.25	2.84	79.6
1	4.02	0.57	6.57	79.6
1	4.02	1.12	10.5	79.6
1	4.02	2.02	9.66	79.6
1	4.02	3.82	8.58	79.6

dplyr joins

```
<join>(x_df, y_df)
```

```
idtime <- data.frame(expand.grid(
       ID = as.numeric(1:3),
       TIME = c(0,1))
) %>% arrange(ID)
```

ID	TIME
1	0
1	1
2	0
2	1
3	0
3	1

ID	WT
1	70
2	80
4	75

```
idwt <- data.frame(
       ID = c(1, 2, 4),
       WT = c(70, 80, 75))
```

METRUM RESEARCH GROUP

joins

- The _join() functions add columns from one data frame to another, matching rows based on the values in common columns.
- The by=c(names) argument is used to explicitly specify the match columns

Function	Description
inner_join(x,y)	Includes all rows that occur in both x and y
left_join(x,y)	Includes all rows that occur in x
right_join(x,y)	Includes all rows that occur in y
full_join(x,y)	Includes all rows that occur in either x or y

INNER JOIN

idtime/idwt => in both

ID	TIME
1	0
1	1
2	0
2	1
3	0
3	1

idtime

ID	WT
1	70
2	80
4	75

idwt

inner_join(idtime, idwt)

ID	TIME	WT
1	0	70
1	1	70
2	0	80
2	1	80

ID	WT	TIME
1	70	0
1	70	1
2	80	0
2	80	1

inner_join(idwt, idtime)

LEFT JOIN

idtime/idwt => in both

ID	TIME
1	0
1	1
2	0
2	1
3	0
3	1

idtime

ID	WT
1	70
2	80
4	75

idwt

left_join(idtime, idwt)

ID	TIME	WT
1	0	70
1	1	70
2	0	80
2	1	80
3	0	NA
3	1	NA

left_join(idwt, idtime)

ID	WT	TIME
1	70	0
1	70	1
2	80	0
2	80	1
4	75	NA

METRUM RESEARCH GROUP

FULL JOIN

idtime/idwt => in both

ID	TIME
1	0
1	1
2	0
2	1
3	0
3	1

idtime

ID	WT
1	70
2	80
4	75

idwt

full_join(idtime, idwt)

ID	TIME	WT
1	0	70
1	1	70
2	0	80
2	1	80
3	0	NA
3	1	NA
4	NA	75

ID	WT	TIME
1	70	0
1	70	1
2	80	0
2	80	1
3	NA	0
3	NA	1
4	75	NA

full_join(idwt, idtime)

METRUM RESEARCH GROUP

ANTI JOIN

idtime/idwt => in both

ID	TIME
1	0
1	1
2	0
2	1
3	0
3	1

idtime

ID	WT
1	70
2	80
4	75

idwt

anti_join(idtime, idwt)

ID	TIME
3	0
3	1

ID	WT
4	75

anti_join(idwt, idtime)

```
pivot_<direction>(df)
```

pivots

- The pivot_longer() function “pivots” the data to make it longer, with more rows and fewer columns
  - The values from multiple columns are placed in one column, with another column containing the previous variable names
  - Often useful for ”stacking” data prior to plotting, when you want to colorize plots by column
- The pivot_wider() function “pivots” the data to make it wider, with fewer rows and more columns
  - Pulls values from one column and the associated names from another to create additional columns
- replaces “gather” and “spread” syntax as of tidyr 1.0.0

PIVOT WIDER

Useful for converting from SDTM/ADaM-style data set to a NONMEM-style data set

ID	name	value
1	SEX	M
1	AGE	45
1	WT	77.2
4	SEX	M
4	AGE	36
4	WT	72.3
7	SEX	F
7	AGE	64
7	WT	119

```
df %>%
pivot_wider(id_cols=ID,
names_from = name,
values_from = value)
```

ID	SEX	AGE	WT
1	M	45	77.2
4	M	36	72.3
7	F	64	119

METRUM RESEARCH GROUP

PIVOT LONGER

ID	SEX	AGE	WT
1	M	45	77.2
4	M	36	72.3
7	F	64	119

```
df %>%
pivot_longer(cols=SEX:WT,
    names_to = "name",
   values_to = "value")
```

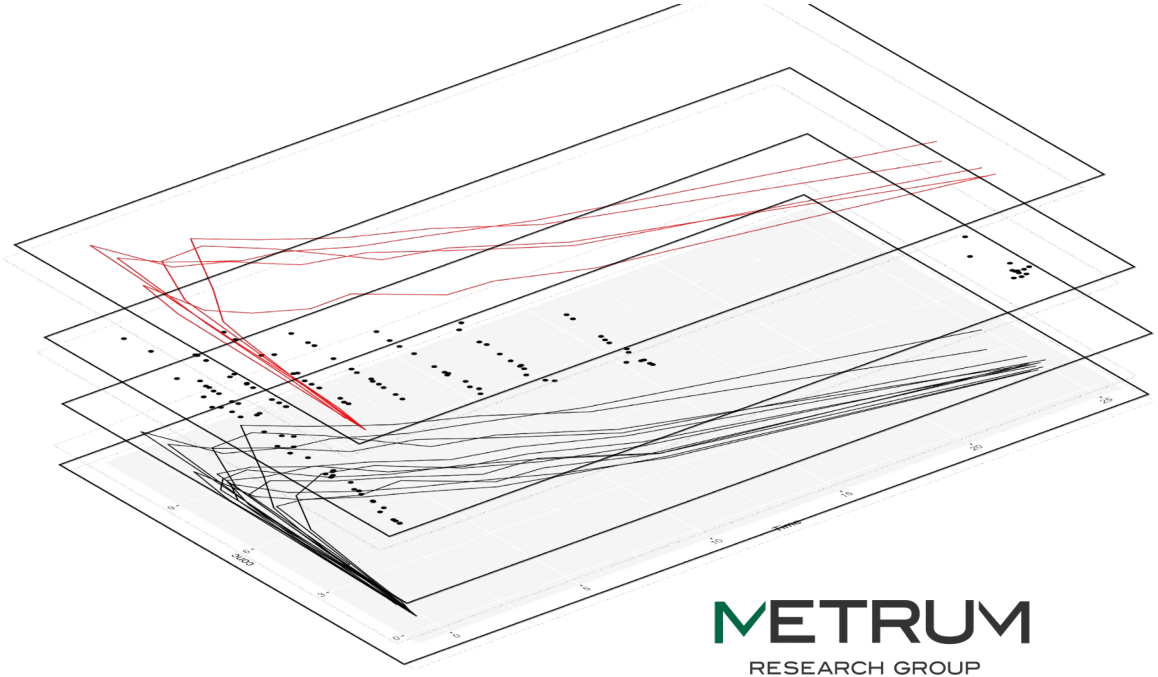
ID	name	value
1	SEX	M
1	AGE	45
1	WT	77.2
4	SEX	M
4	AGE	36
4	WT	72.3
7	SEX	F
7	AGE	64
7	WT	119

METRUM
RESEARCH GROUP

ggplot

ggplot: a modern system for data visualization

- The grammar of graphics (gg) consists of the sum of several independent layers of objects that are combined using + to construct the final graph



Layers are stacked sequentially to create the final plot

```
ggplot(aes()) +
    geom_line() +
    geom_point() +
    geom_line(col="red")
```

METRUM
RESEARCH GROUP

Elements of ggplot

Element	Code	Use
data	data =	Raw data for plotting
aesthetics:	aes()	Aesthetics of the geoms and stats – control **color, size, shape, fill**, etc and how it is mapped to underlying data
geometries:	geom_<type>	Shapes to represent the data +settings unrelated to data
scales	scale_	Maps the coordinate system for the geoms
Statistical transformation	stat_	transforms the data, typically by summarizing it in some manner.
Additional Customizations	theme	Control axis, background, tick settings (size, color, etc)

https://rstudio.github.io/cheatsheets/

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.

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

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

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

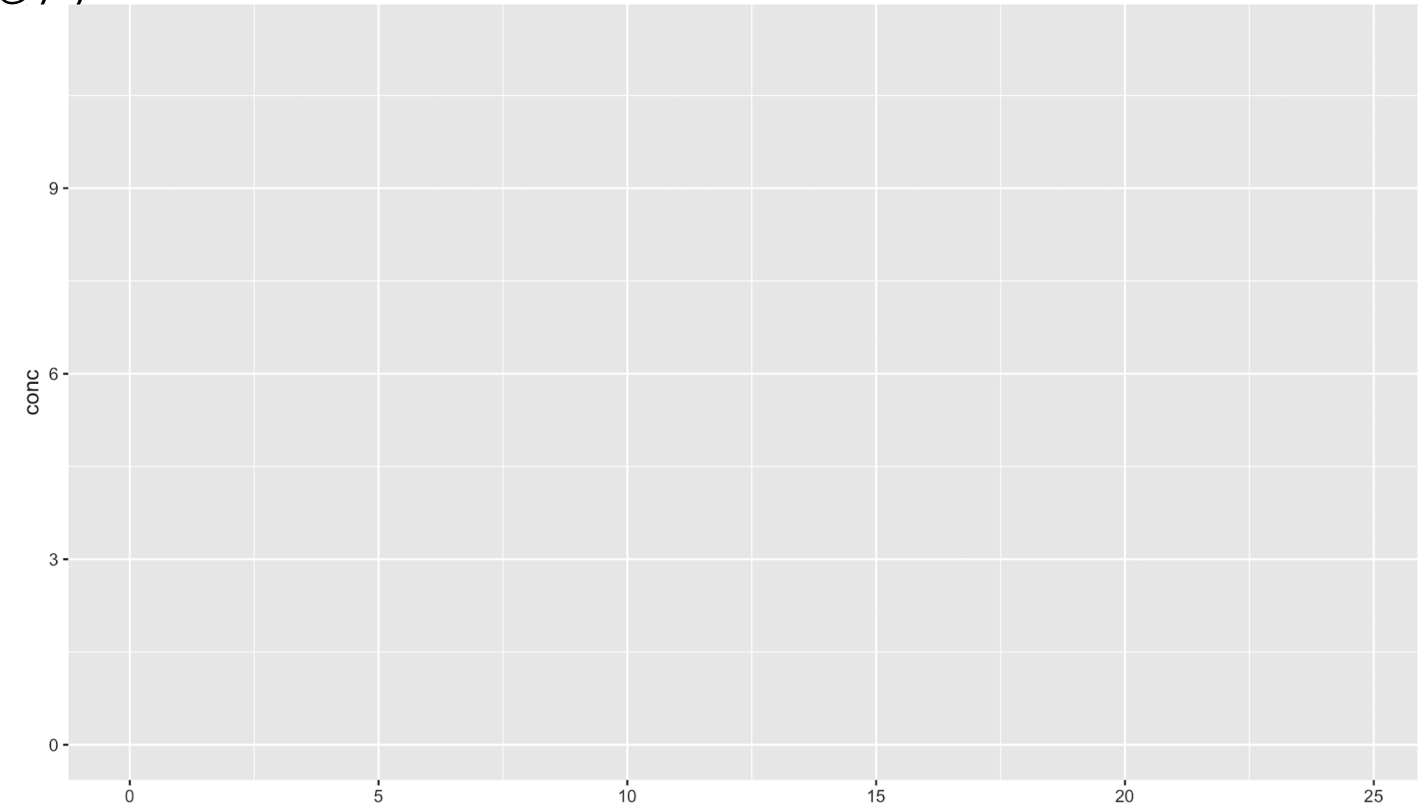
```
ggplot(data = Theoph, aes(x = Time, y = conc))
```

Underlying data

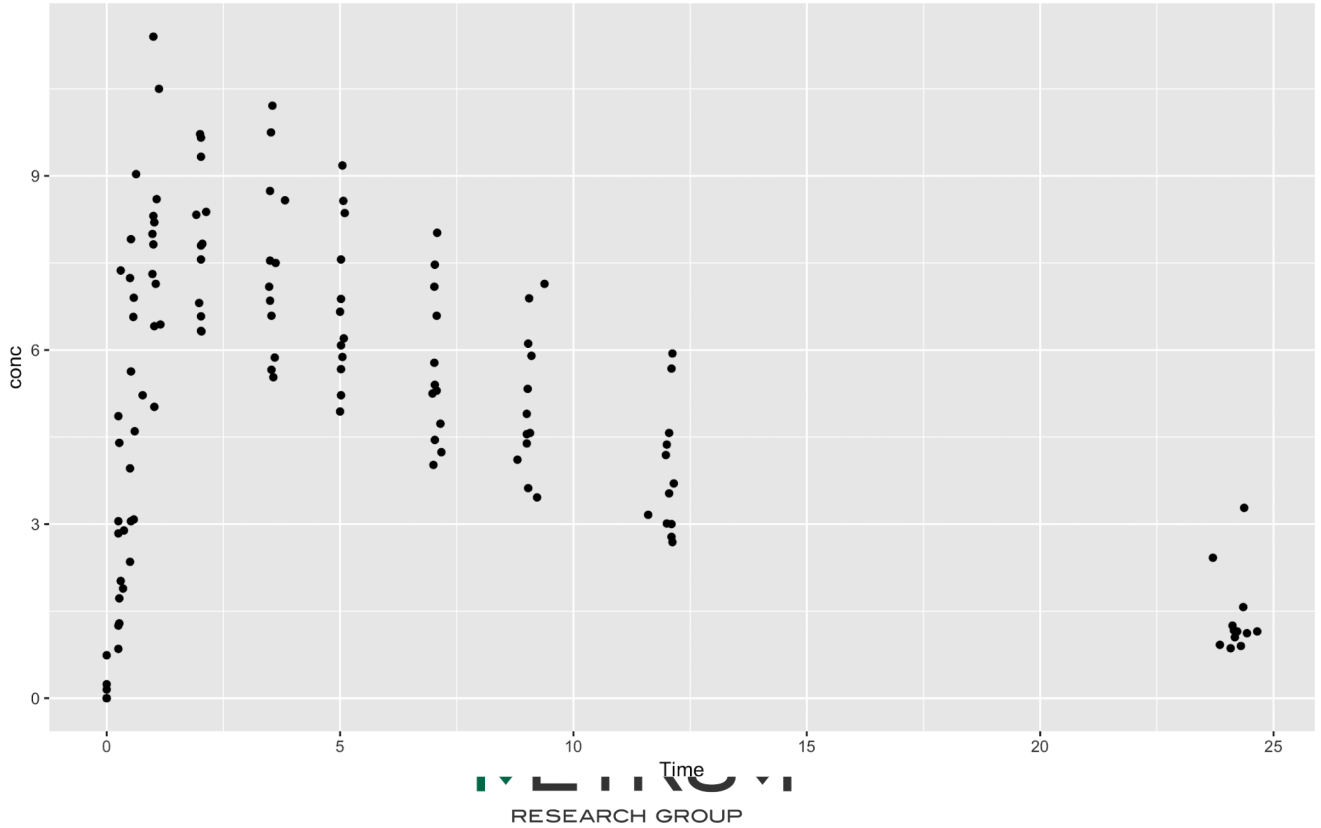
Data columns and their associated aesthetic ‘mappings’

Subject	Wt	Dose	Time	conc
1	79.6	4.02	0	0.74
1	79.6	4.02	0.25	2.84
1	79.6	4.02	0.57	6.57
1	79.6	4.02	1.12	10.5
1	79.6	4.02	2.02	9.66
1	79.6	4.02	3.82	8.58

```
ggplot(data=Theoph, aes(x = Time, y = conc))
```

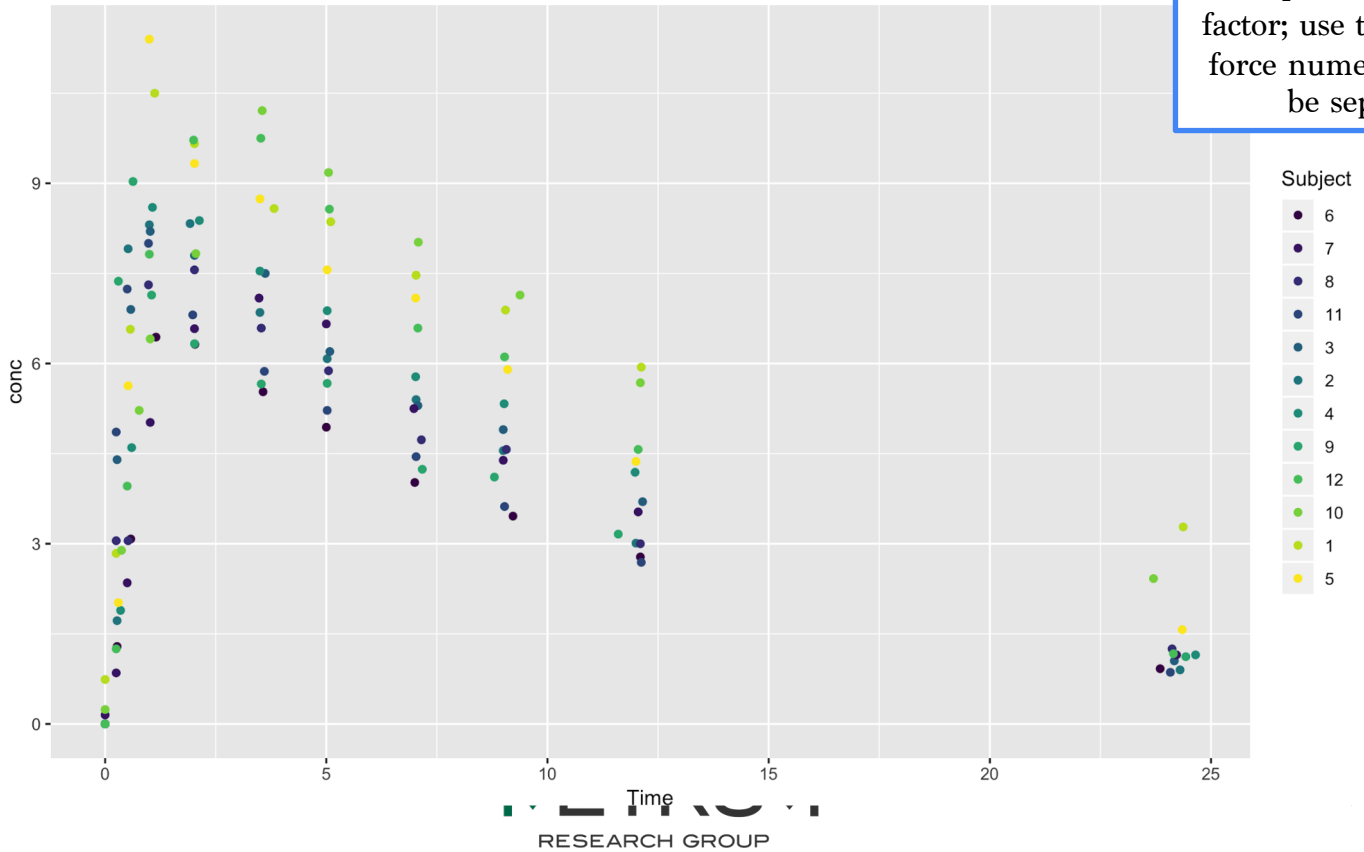


```
ggplot(data=Theoph, aes(x = Time, y = conc)) +
    geom_point()
```



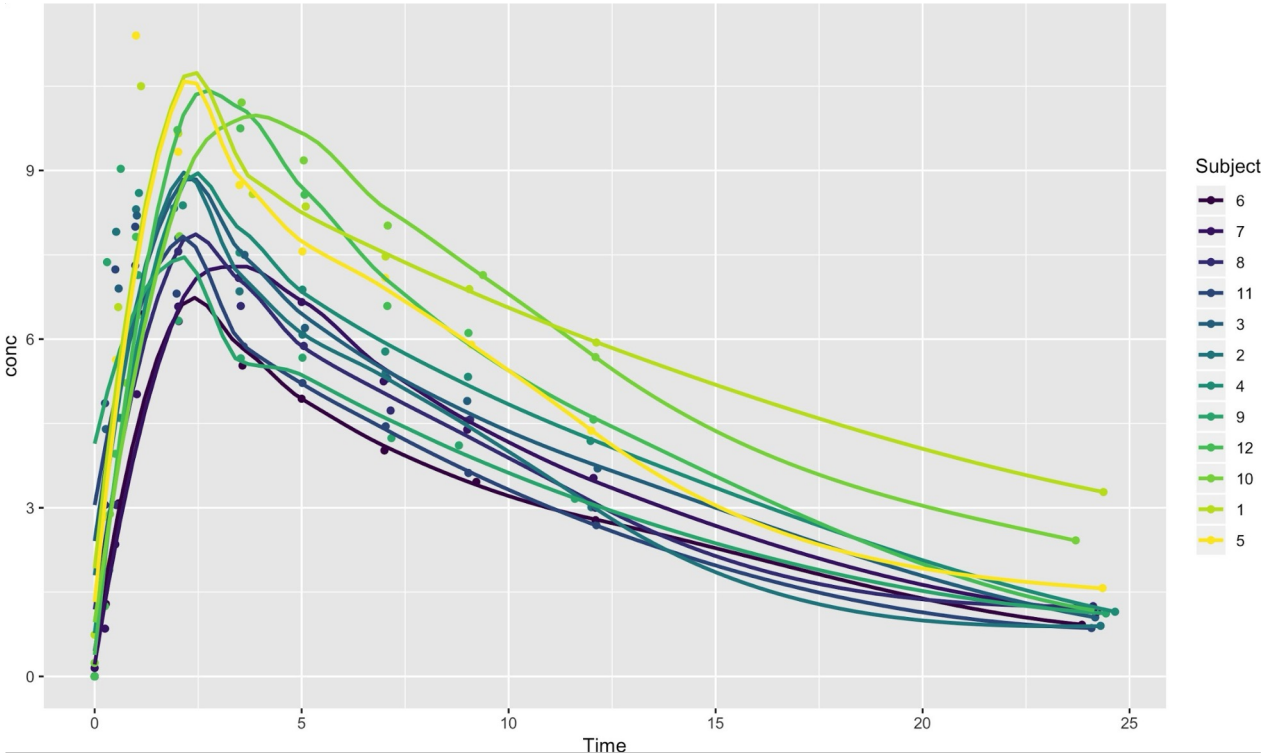
```
ggplot(data=Theoph, aes(x = Time, y = conc,
color=Subject)) + geom_point()
```

`Theoph$Subject` is a factor; use typecasting to force numeric values to be separated



RESEARCH GROUP

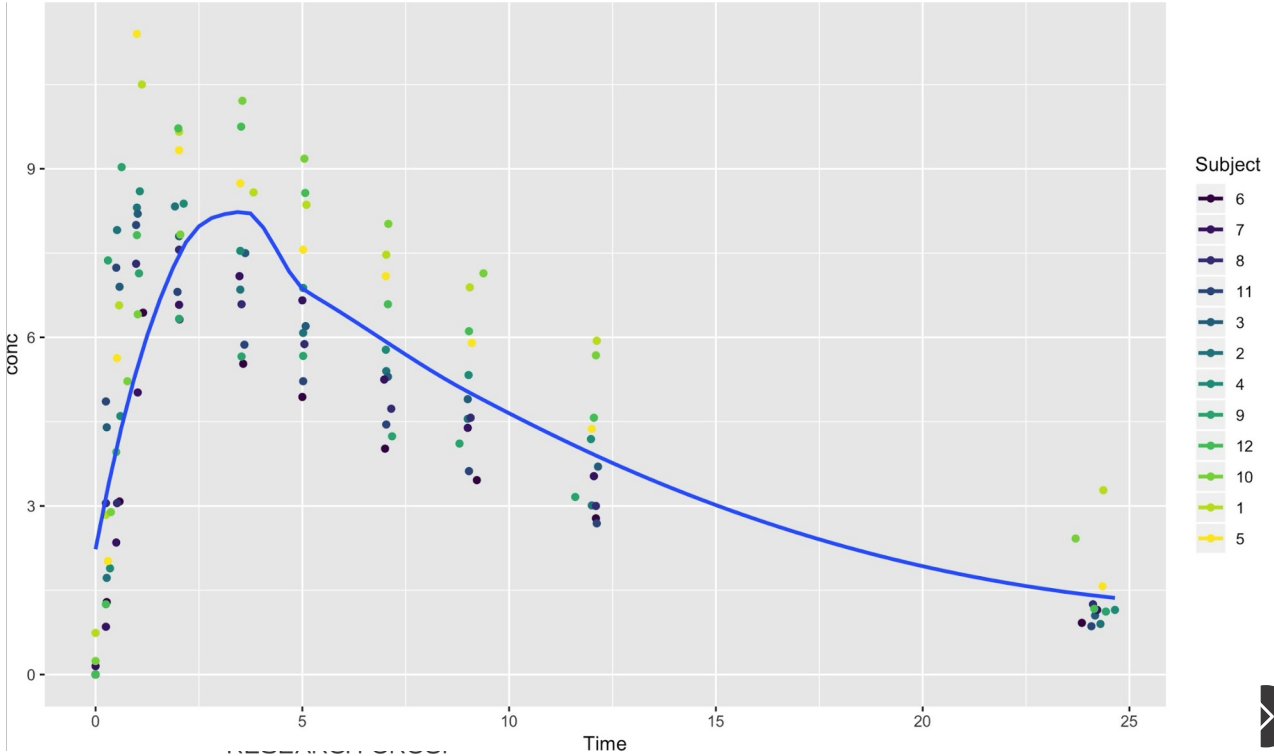
```
ggplot(data=Theoph,aes(x=Time, y=conc,
color=Subject))+ geom_point()+ geom_smooth(se=FALSE)
```



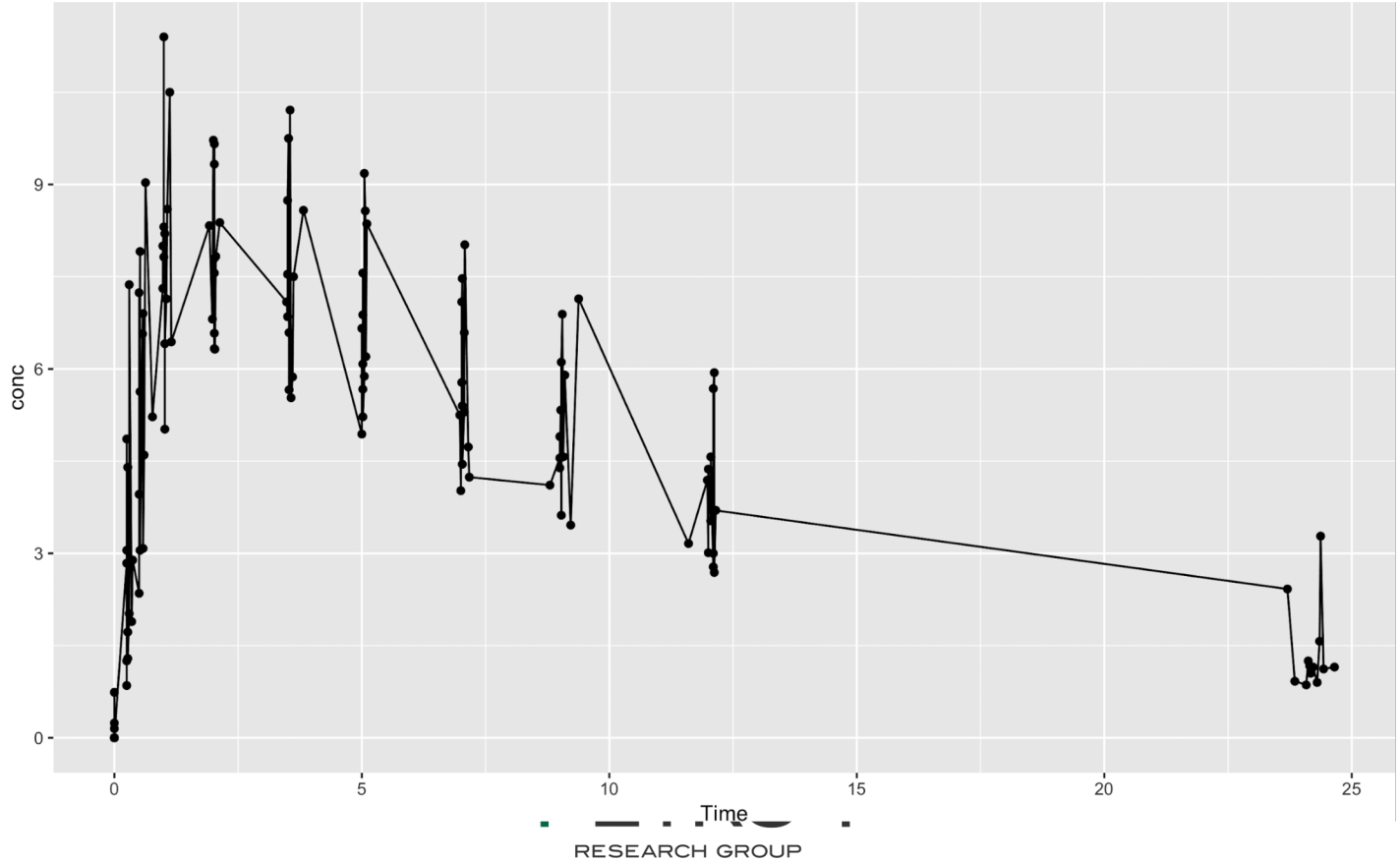
Propagating aesthetic properties


```
ggplot(data=Theoph,aes(x=Time,y=conc, color = Subject))+
  geom_point()+
  geom_smooth(se=FALSE, aes(color = NULL))
```

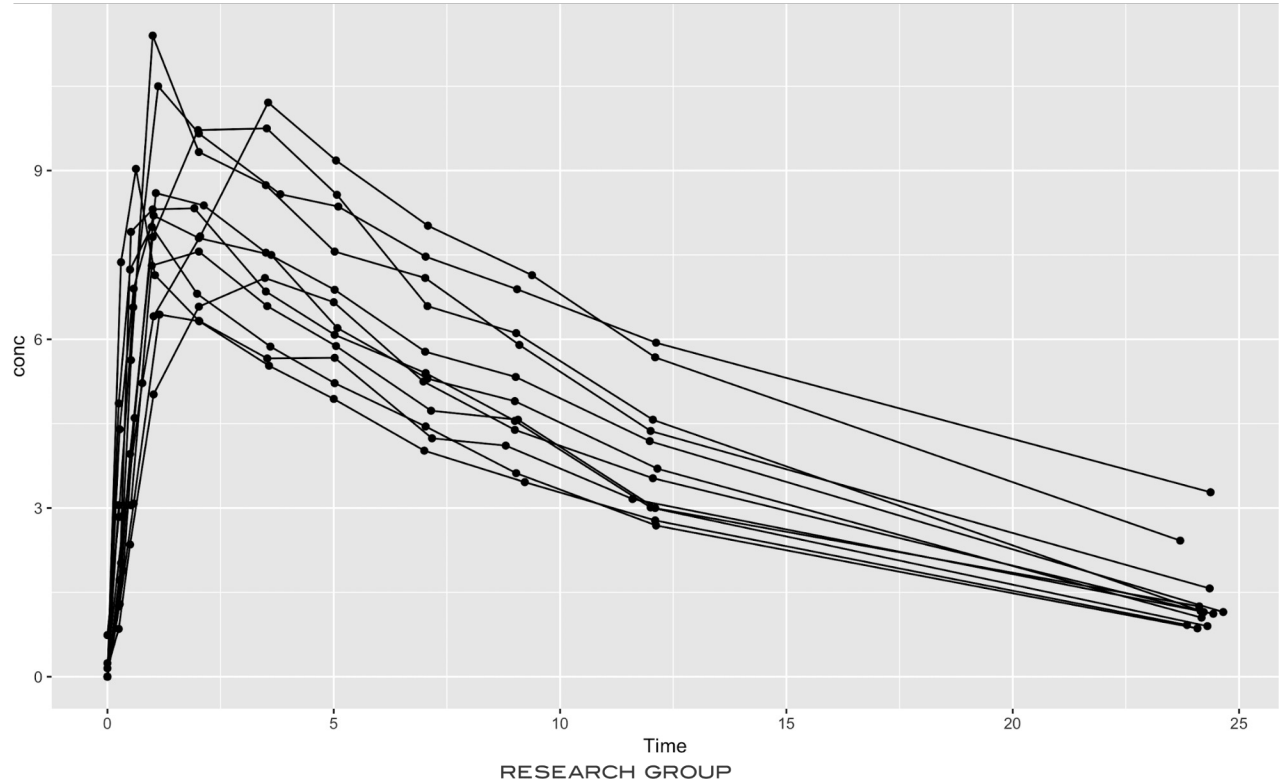
data and aes() are inherited from ggplot call, but can also be assigned individually for each geom



```
ggplot(data=Theoph, aes(x = Time, y = conc)) +
geom_point() + geom_line()
```



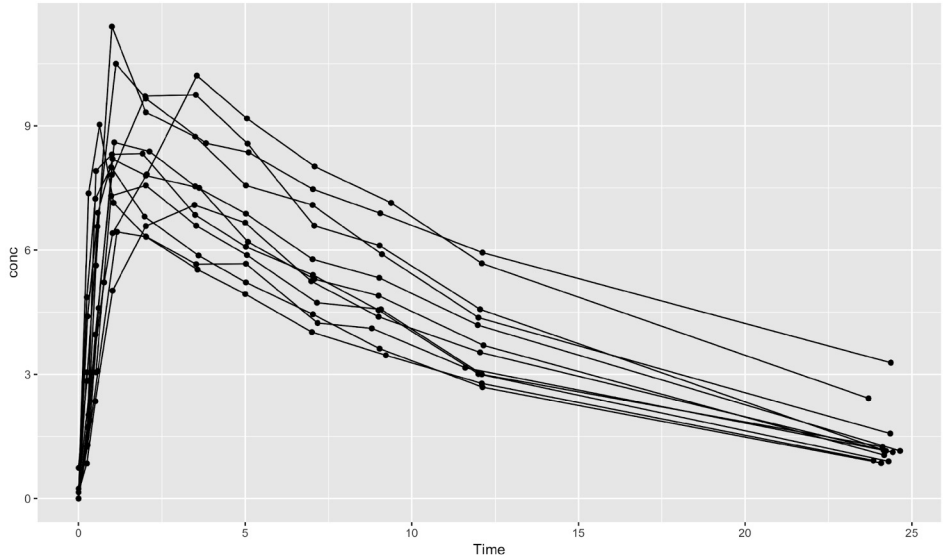
```
ggplot(data=Theoph, aes(x = Time, y = conc,
    group = Subject)) +
    geom_point() + geom_line()
```



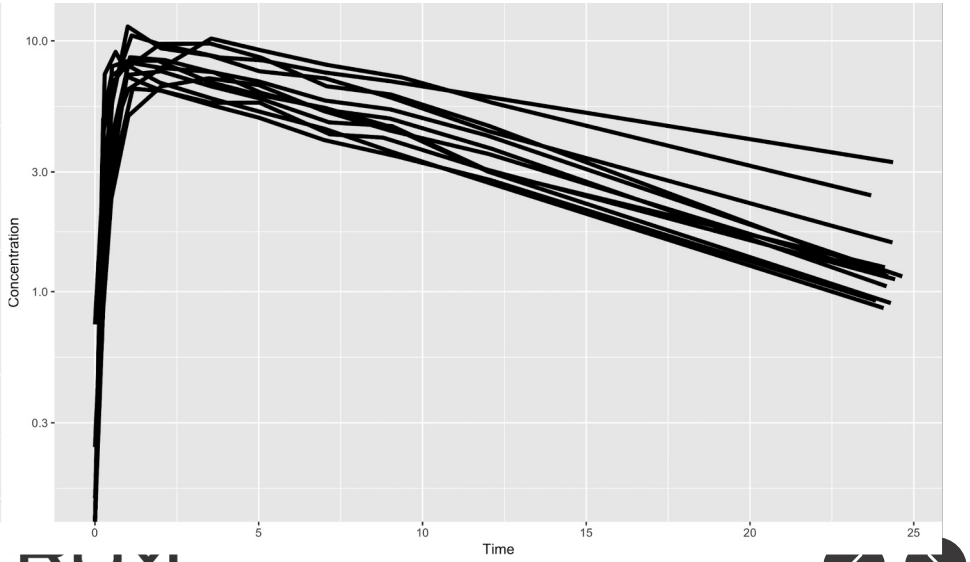
Objects can be saved and more layers added


```
conc_time <- ggplot(data = Theoph,
    aes(x = Time, y = conc, group = Subject)
) + geom_line()
```

conc_time



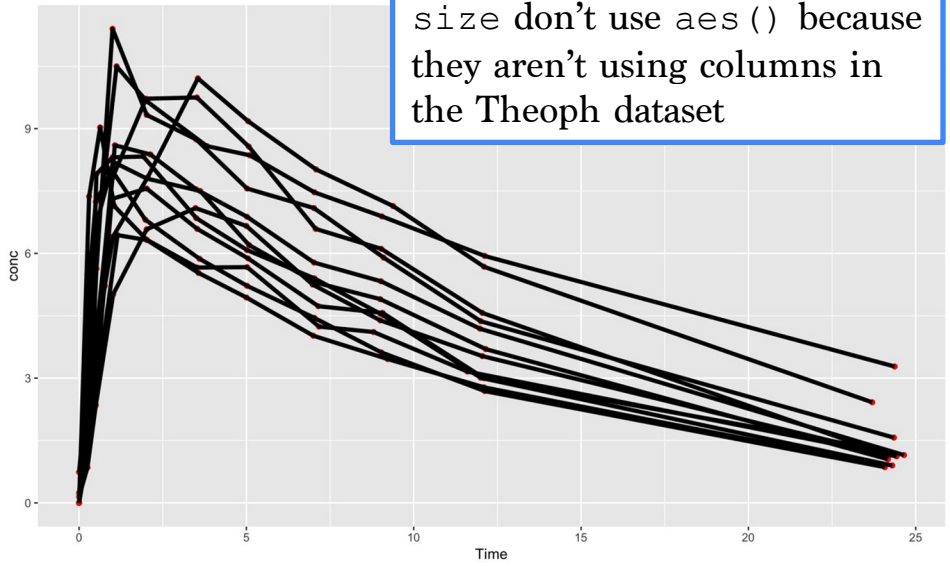

```
conc_time +
  scale_y_log10() +
  ylab("Concentration")
```



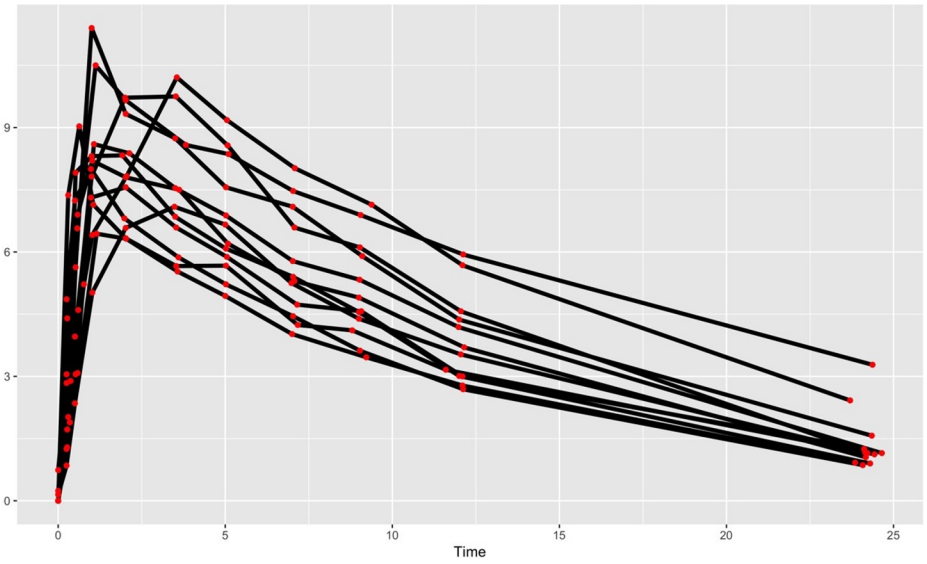
Order matters visually!

```
ggplot(data = Theoph, aes(x = Time,
  y = conc, group = Subject)) +
  geom_point(color = 'red') +
  geom_line(size = 1.5)
```

Notice here that `color` and `size` don't use `aes()` because they aren't using columns in the Theoph dataset



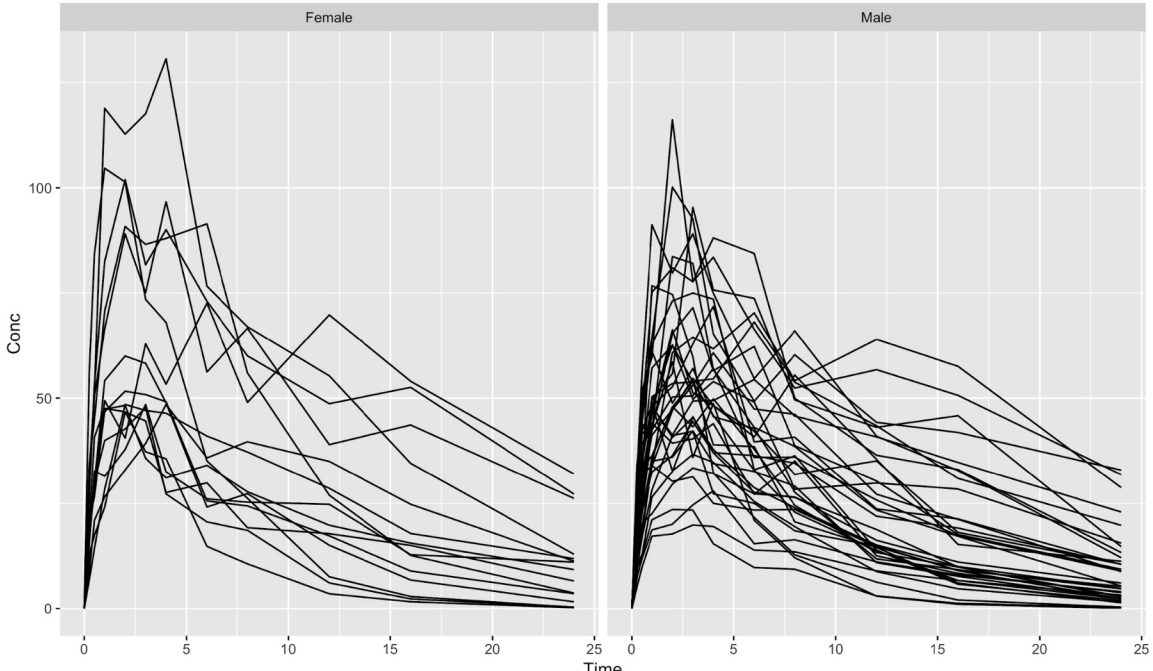

```
ggplot(data = Theoph, aes(x = Time,
  y = conc, group = Subject)) +
  geom_line(size = 1.5) +
  geom_point(color = 'red')
```



Facets


```
ggplot(sd_oral_richpk, aes(x = Time, y = Conc)) +
    geom_line(aes(group = ID)) +
    facet_wrap(~Gender)
```

`facet_grid()` also exists which facets by more than one variable



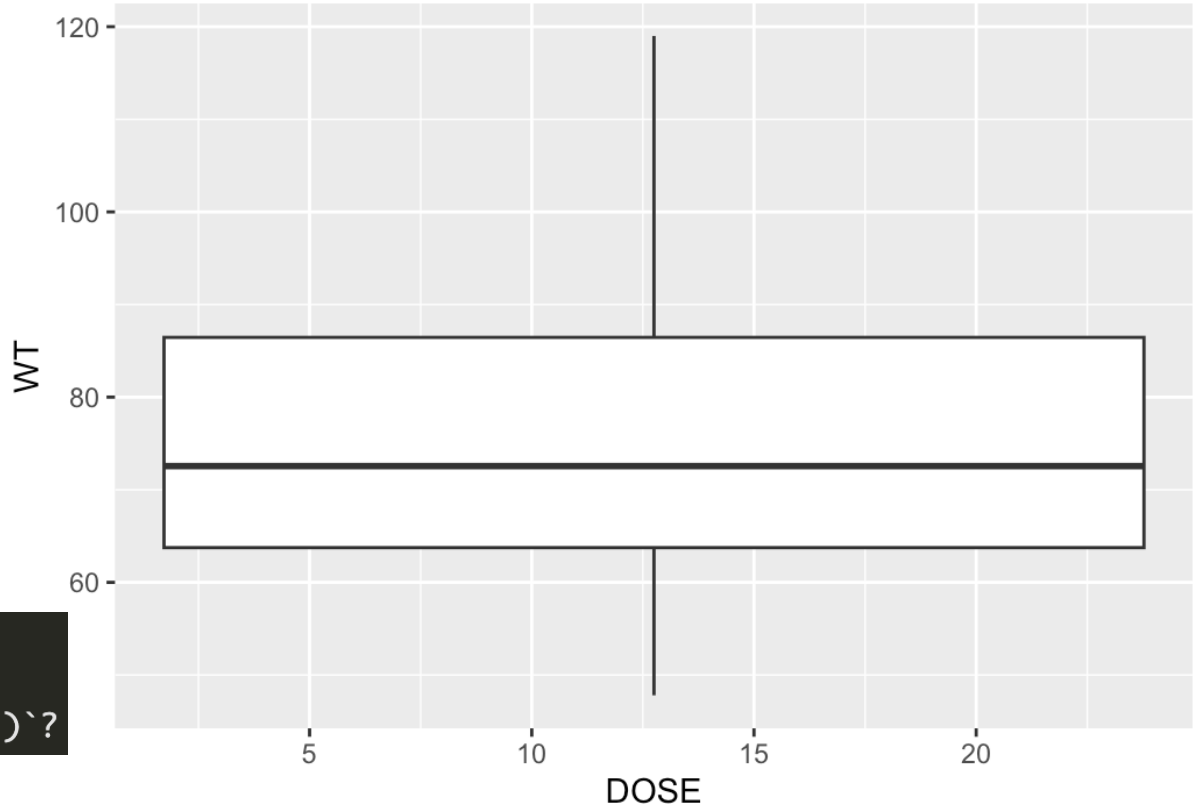
Facets are ordered alphabetically unless they are factors – use factors to enforce ordering of facets

boxplots in ggplot

```
dat %>%
group_by(ID) %>% slice(1) %>% ungroup() %>%
  ggplot(aes(x=DOSE, y=WT)) +
    geom_boxplot()
```

This is not what we want.

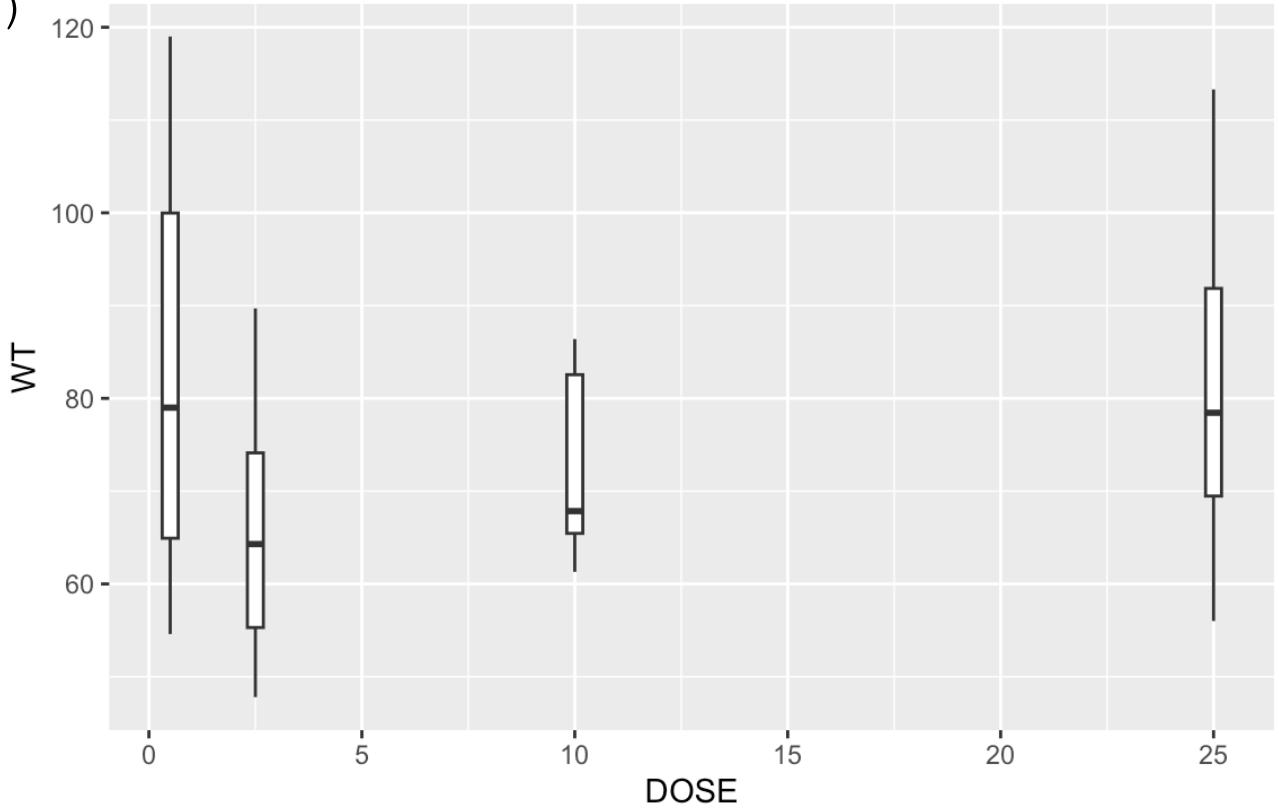
Creating boxplots requires the `group` element to be defined to work properly




```
Warning message:
Continuous x aesthetic
i did you forget `aes(group = ...)`?
```

```
dat %>%
  group_by(ID) %>% slice(1) %>% ungroup() %>%
  ggplot(aes(x=DOSE, y=WT, group=DOSE)) +
    geom_boxplot()
```

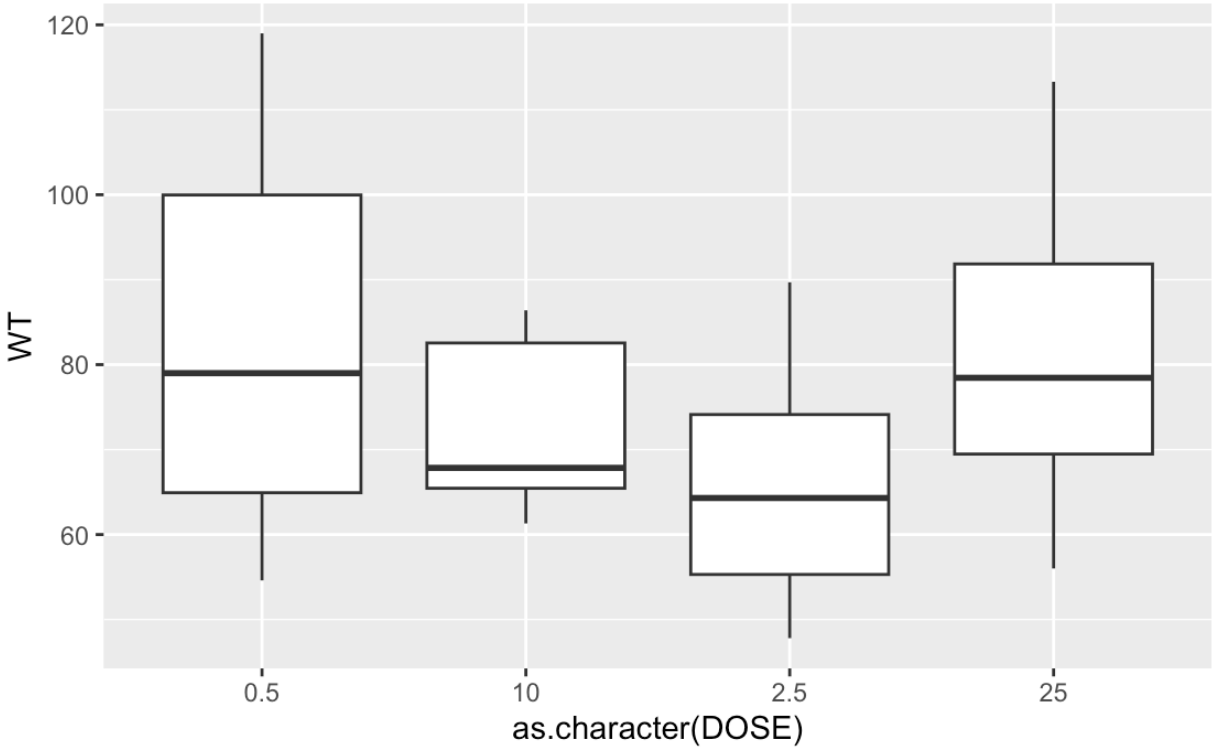
This is getting closer, but why are we seeing the weird spacing on the x-axis?



```
dat %>%
  group_by(ID) %>% slice(1) %>% ungroup() %>%
  ggplot(aes(x=as.character(DOSE),
             y=WT, group=as.character(DOSE))) +
    geom_boxplot()
```

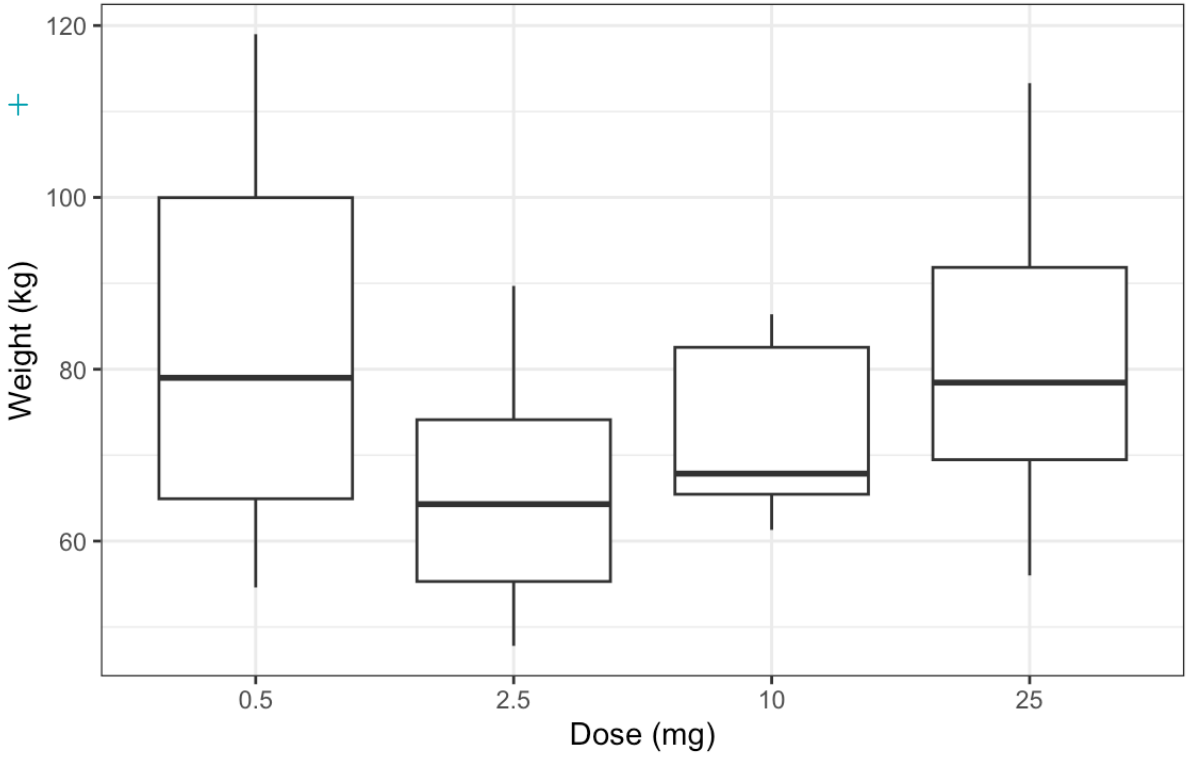
`group` is no longer required when x is a character/factor

Our spacing is fixed, but the order is not correct because it's alphabetical

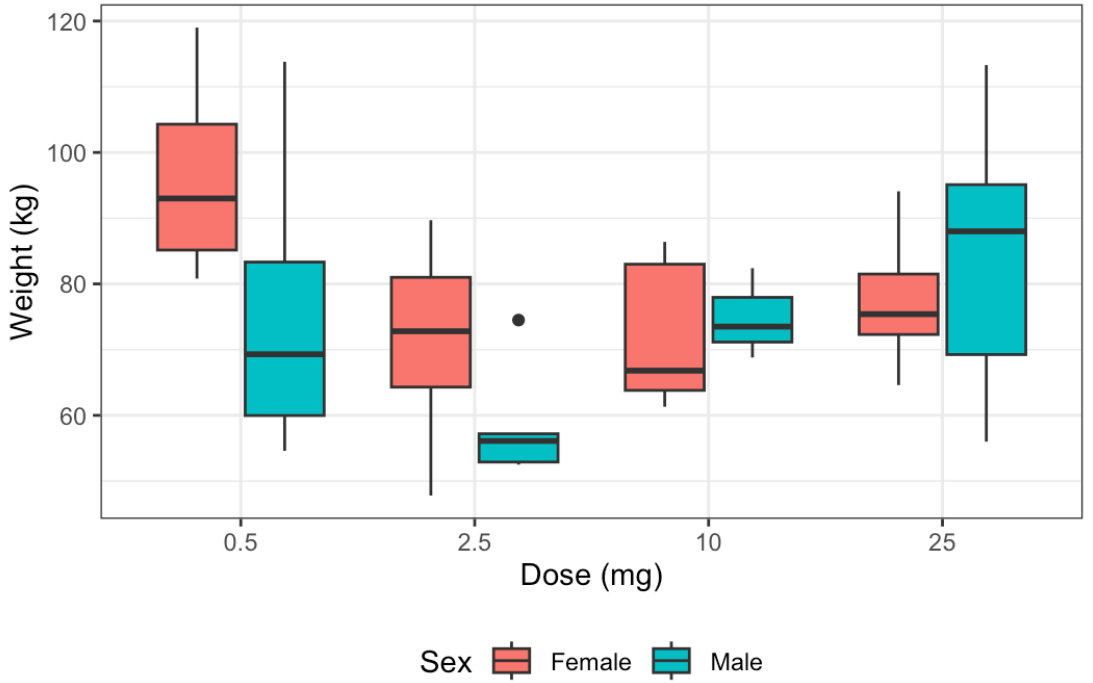


```
dat %>%
  group_by(ID) %>% slice(1) %>% ungroup() %>%
  mutate(DOSE_f = factor(DOSE, levels = c(0.5,2.5,10,25))) %>%
  ggplot(aes(x=DOSE_f, y=WT)) +
    geom_boxplot() +
    labs(x="Dose (mg)",
         y="Weight (kg)") +
    theme_bw()
```

Creating factors enforces order for faceting as well!

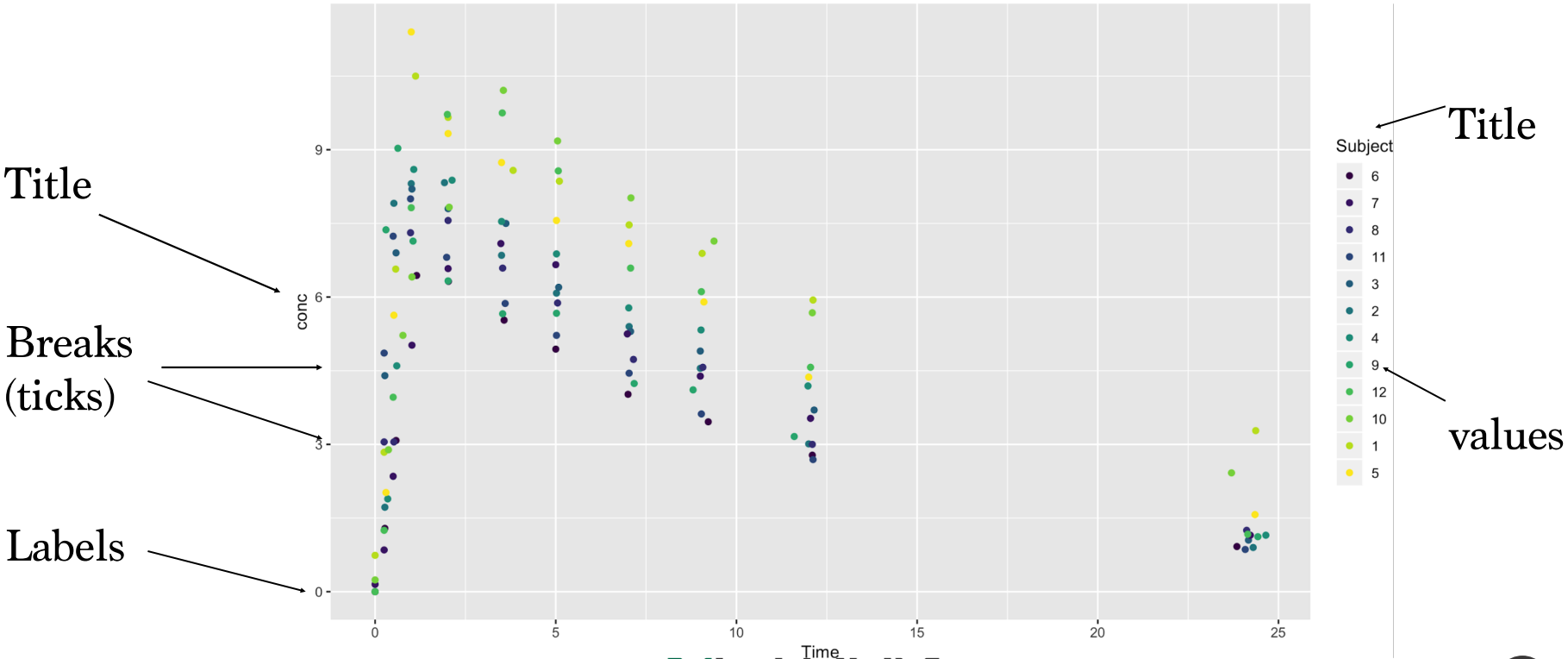


```
dat %>%
 <slice and mutate> %>%
  ggplot(aes(x=DOSE_f, y=WT, fill=SEX_c)) +
    geom_boxplot() +
    labs(x="Dose (mg)", y="Weight (kg)", fill="Sex") +
    theme_bw() + theme(legend.position = ”bottom")
```



customizing aesthetics

Scales relate to axis and legends



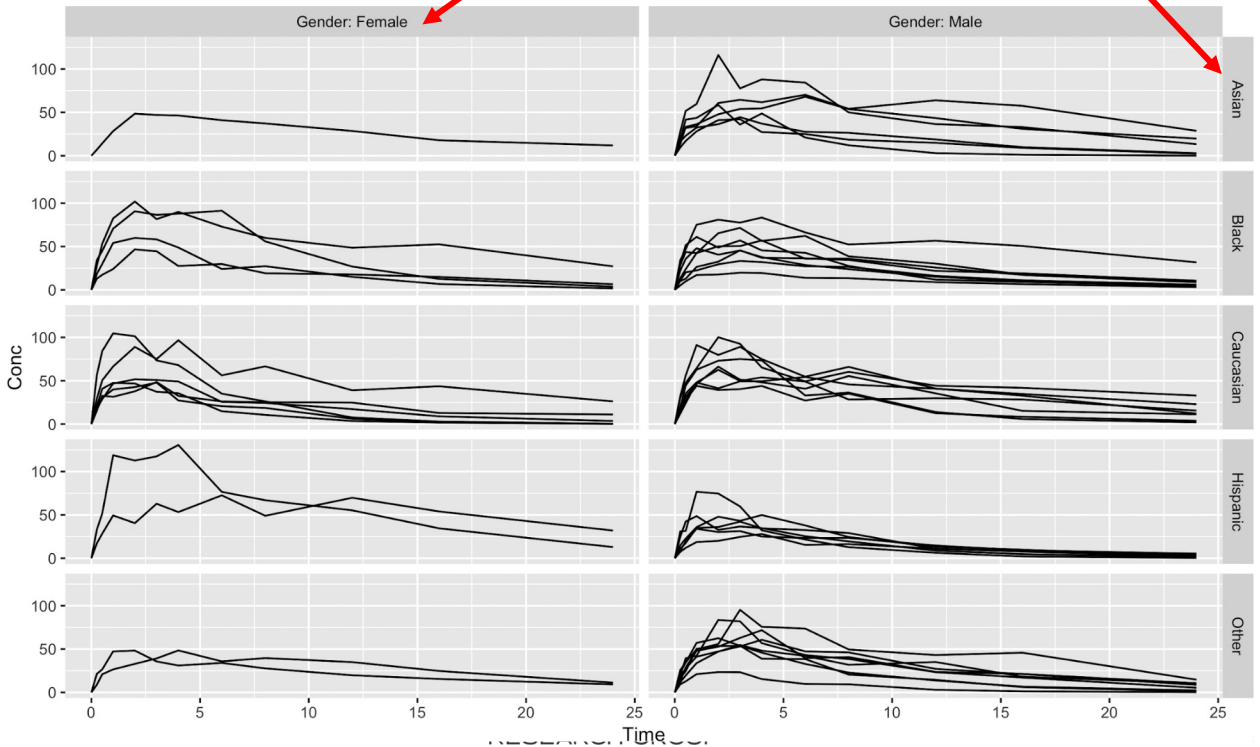
scale_<aes>_<type>(name, breaks, labels)


```
ggplot(Theoph, aes(time, conc)) +
geom_point(aes(color = Subject)) +
scale_color_discrete(name = 'ID') +
scale_y_continuous(name = 'Concentration (mg/L)',
                   breaks = c(0, 0.5, 1, 2, 5, 10))
```

Naming convention for a scale has three elements separated by "_"

1. scale
2. The name of the aesthetic (e.g., color, shape or etc)
3. The name of the scale (e.g., continuous, discrete, manual).

```
ggplot(sd_oral_richpk, aes(x = Time, y = Conc)) +
geom_line(aes(group = ID)) +
facet_grid(Race~Gender,
labeller = labeller(Gender = label_both, Race = label_value))
```



Summary of ggplot

- Objects are layered in sequential order to create the plot
  - first line = bottom layer, last line = top layer
- Aesthetics and data are inherited from ggplot() object, but can be overwritten within individual geometries (`geom_*`)
- Grouping is important to ensure plots turn out how you want them to look
  - adding aesthetics enforces grouping as well
- There are a lot of helpful tutorials on the internet, don't hesitate to Google (or ask ChatGPT) how to make a specific plot

Loops and Functions

Don’t copy-paste your code

- Whenever you get the urge to copy a chunk of code and use it somewhere else in the same script, you should write a function
  - Prevents accidental typos introduced in copy-pasting
  - Anything you would change becomes an argument

- “When will I use this?”
  - Helper functions
  - Repetitive workflows
  - Simulations/bootstraps
  - Plotting

Anatomy of a function in R

```
## function to calculate factorial of x
my_factorial <- function(x){
  ## calculate product for all numbers from 1 to x
  product <- prod(seq(1,x,1))
  ## return product
  return(product)
}

## function test
my_factorial(3)
> 6
```

Function name
Function arguments
Internally-scoped variables
Return statement

A two-argument example

```
## function to implement the "choose" operator
my_choose <- function(x,y){
  ## calculate x choose y using our factorial function
  my_factorial(x) / (my_factorial(y) * my_factorial(x-y))
}

## test function
my_choose(3,2)
> 3
```

If `return()` is not explicitly called, the outputs of the last line will be returned automatically

BUT it is better to be explicit!

Using defaults and ellipsis

```
ci95 <- function(x, lo_q=0.025, hi_q = 0.975, ...){
  lo <- quantile(x, probs=lo_q, ...)
  hi <- quantile(x, probs=hi_q, ...)
  return(c(lo, hi))
}
ci95(1:200)
>  2.5% 97.5%
>  5.975 195.025

ci95(1:200, names=FALSE)
>  5.975 195.025
```

Using function outputs

```
## assign function output to named variable
out_90 <- ci95(1:200, lo_q=0.05, hi_q = 0.95)

out_90
>    5%    95%
> 10.95 190.05

## display output in meaningful context
paste("90% CI:", out_90[1], "-", out_90[2])
> "90% CI: 10.95 - 190.05"
```

Summary of functions

- If your function shares a name with a function already in the environment, the existing function will be overwritten with your new function
  - Bad coding practice to use existing function names
- Arguments without default values are required for the function to run
- Arguments with defaults will use the default value unless the function call provides new values
- Ellipsis are used to pass additional arguments to calculations inside a function
- Any object created inside a function cannot be accessed outside of the function (this is called “scoping”)
- Comments and meaningful variable names are the best way to ensure your function can be reused by other people (and future you!)

The “for” loop

- In programming, a “loop” means a process which is repeated a certain number of times (“for” loop) or until a specific condition is met (“while” loop)
  - We won’t discuss while loops today since they are not used as frequently in data analysis

loop counter variable

```
for(i in 1:n){<operation>}
```

function call

loop sequence

A simple for loop

```
print("ignition sequence")

for (i in 10:1){
  print(i)
}

print("blastoff")
```

Executing code in a for loop

```
vec <- c( "one", "two", "three" )

for ( i in 1:length( vec ) ) {
    if ( vec[ i ] == "one" ) {
        vec[ i ] <- "1"
    }
}
message( "Replaced vector:" )
print( vec )
```

here is an example of a conditional statement which differs from the `if_else()` format shown earlier

Vector calculations in R

- For loops are common in most programming languages
- R is built for vector calculations, making many for loops unnecessary
- The code on the previous page could be re-written without a loop as such:

```
vec <- c( "one", "two", "three" )
vec [ vec == "one" ] <- "1"
print( vec )
```

apply() functions

- Furthermore, the for loop is slow in R due to its emphasis on vectorized coding
- Base R has a series of functions called the apply() functions which are recommended to be used in place of for loops
  - apply(), lapply(), sapply(), tapply(), vapply()
- These functions do the same thing as a for loop, but run much faster

map: how tidyverse implements for loops

- The purrr package implements many looping mechanism, most importantly the map function

loop sequence

```
map(1:n,function(i)<operation>)
```

function call

loop counter variable

Using functions in map

```
vec <- c("one", "two", "three")
replace_one <- function(i) if_else( i == "one" , "1", i)
purrr::map(vec, replace_one)
[[1]]
[1] "1"

[[2]]
[1] "two"

[[3]]
[1] "three"
```

map defaults to returning output as a list, but there are several helper functions which will automatically typecast outputs

Typecasting output and anonymous functions

```
> out <- purrr::map_chr(vec, function(i) {
+   if_else( i == "one" , "1", i)
+ })
> class(out)
[1] "character"
> out
[1] "1"     "two"   "three"
```

Simulating a small clinical trial population

```
sim_trial_arm <- function(x, n_subjects=4) {
  data.frame(
    ARM = x,
    ID = seq(n_subjects),
    DOSE = <some function>,
    MALE = <some function>,
    WT = <some function>
  ) %>% return()
}
```

Simulating a single arm

```
> sim_trial_arm(1)
  ARM ID     DOSE MALE    WT
    1  1    10 mg    1 100.3
    1  2    10 mg    1  94.7
    1  3    25 mg    0  83.3
    1  4 Placebo    0  67.4
```

Simulating multiple arms using map()

```
> map_df(seq(3), sim_trial_arm)
   ARM ID    DOSE MALE   WT
1    1  1   25 mg    1 84.6
2    1  2 Placebo    1 74.0
3    1  3   10 mg    1 76.0
4    1  4   25 mg    0 67.8
5    2  1   10 mg    1 89.5
6    2  2   10 mg    1 78.2
7    2  3 Placebo    1 77.3
8    2  4   25 mg    1 69.2
9    3  1   25 mg    1 65.4
10   3  2   25 mg    0 82.2
11   3  3 Placebo    0 78.3
12   3  4   25 mg    0 80.1
```

```
> map_df(seq(3), sim_trial_arm)
   ARM ID    DOSE MALE    WT
1    1  1   25 mg    0  69.2
2    1  2   25 mg    0  88.0
3    1  3 Placebo    0  83.6
4    1  4 Placebo    1  80.9
5    2  1   25 mg    1  96.3
6    2  2   10 mg    0  72.8
7    2  3 Placebo    0 106.2
8    2  4   10 mg    1  98.4
9    3  1 Placebo    1  75.8
10   3  2   10 mg    1  68.6
11   3  3   25 mg    0  79.5
12   3  4   25 mg    1  70.6
```

But, our results aren't reproducible!

An aside on random numbers in R

- A random number generator is used for any function in R that requires some amount of stochasticity (e.g. sampling from a distribution)
- The `set.seed()` function is one way to set an initial value in the random number generator so that reproducible results can be obtained
- Examples of commonly used random functions in R
  - Sample from uniform distribution: `runif(n=5, min=0, max=1)`
  - Sample from normal distribution: `rnorm(n=5, mean=0, sd=1)`
  - Sample from an object in R: `sample(c("red", "green"), x=5, replace=TRUE)`

Simulating a small clinical trial population

```
sim_trial_arm <- function(x, n_subjects=4) {
  data.frame(
    ARM = x,
    ID = seq(n_subjects),
    DOSE = sample(c("10 mg", "25 mg", "Placebo"),
                  size=n_subjects, replace=TRUE),
    MALE = runif(n_subjects) %>% round(),
    WT = rnorm(n_subjects, mean=80, sd=12) %>% round(digits=1)
  )
}
```

Setting RNG seed allows results to be reproduced

```
> set.seed(123)
> map_df(seq(3), sim_trial_arm)
   ARM ID    DOSE MALE    WT
1    1  1 Placebo    1  81.6
2    1  2 Placebo    0 100.6
3    1  3 Placebo    1  85.5
4    1  4   25 mg    1  64.8
5    2  1   10 mg    1  86.6
6    2  2 Placebo    1  82.9
7    2  3 Placebo    1  67.4
8    2  4   10 mg    1  95.5
9    3  1   25 mg    0  77.4
10   3  2   10 mg    0  67.7
11   3  3 Placebo    0  71.3
12   3  4 Placebo    0  72.5
```

```
> set.seed(123)
> map_df(seq(3), sim_trial_arm)
   ARM ID    DOSE MALE    WT
1    1  1 Placebo    1  81.6
2    1  2 Placebo    0 100.6
3    1  3 Placebo    1  85.5
4    1  4   25 mg    1  64.8
5    2  1   10 mg    1  86.6
6    2  2 Placebo    1  82.9
7    2  3 Placebo    1  67.4
8    2  4   10 mg    1  95.5
9    3  1   25 mg    0  77.4
10   3  2   10 mg    0  67.7
11   3  3 Placebo    0  71.3
12   3  4 Placebo    0  72.5
```

METRUM RESEARCH GROUP

How I'm currently ensuring reproducible results

```
res <- withr::with_seed(123,
  map_df(seq(3), sim_trial_arm)
)
```

- Explicitly connects seed to stochastic function call
- If running code interactively, ensures `set.seed()` isn't missed

Adjusting arguments using the tilde operator

```
> purrr::map_df(seq(3), ~ sim_trial_arm(.x, n_subjects = 6))
   ARM ID    DOSE MALE   WT
1    1  1   10 mg    1 90.7
2    1  2 Placebo    0 90.5
3    1  3   10 mg    1 89.9
4    1  4 Placebo    0 88.3
5    1  5   25 mg    0 86.6
6    1  6   10 mg    0 79.3
7    2  1 Placebo    1 75.2
8    2  2   25 mg    1 74.4
9    2  3   25 mg    1 89.4
10   2  4 Placebo    0 79.0
11   2  5   25 mg    0 83.0
12   2  6   25 mg    1 79.7
13   3  1   25 mg    1 84.6
14   3  2   10 mg    0 74.0
15   3  3 Placebo    1 76.0
16   3  4 Placebo    1 67.8
17   3  5   25 mg    1 67.1
18   3  6 Placebo    0 83.6
```

The tilde operator ~ is equivalent to `function(.x)`

METRUM RESEARCH GROUP

Additional Resources

https://intro2r.com/ - Introduction to Base R and RStudio

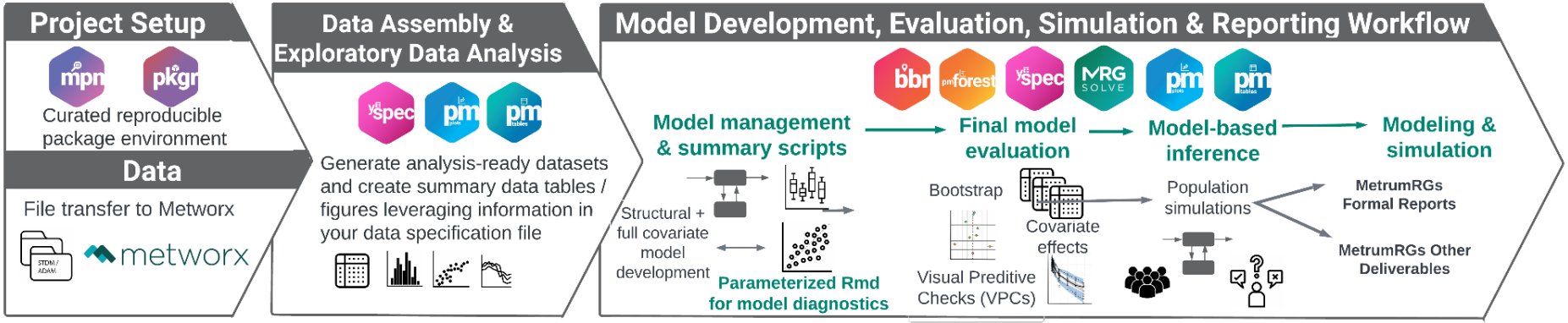
https://dominicroye.github.io/en/2020/a-very-short-introduction-to-tidyverse/
A short introduction to the Tidyverse

https://www.rstudio.com/resources/cheatsheets/
Quick reference guides for tidyverse functions

https://r4ds.had.co.nz/
Free textbook “R for Data Science” by Hadley Wickham covering many topics

Open-source Pharmacometrics R Packages

www.metrumrg.com/merge-expo



- What you'll find in this Expo:
  - Our approach to project set-up, data assembly, M&S activities, and reporting.
  - Access to example code in a Github repository.
  - Information and vignettes on MetrumRG's suite of tools.

METRUM
RESEARCH GROUP

Homework Problems

Problem #1: Building a data.frame

- Create a data.frame matching the table shown
  - hint: type `?rep()` into the console to see help for the replicate function
- Save the output to an object named “dat”
- Verify that the structure and contents of dat are correct
- Save “dat” to disk in your working directory with the name “output.csv”

ID	TIME	DV	SEX	WT	ROW
1	1	2	M	70	1
1	6	4	M	70	2
1	24	6	M	72	3
2	1	2	F	64	4
2	6	4	F	64	5
2	24	6	F	63	6
3	1	2	M	98	7
3	6	4	M	98	8
3	24	6	M	98	9

Problem #2: Modifying an existing data set

- Import mad-nonmem.csv
- Filter out commented rows (i.e. in column C, rows equal to “C”)
- Create new columns in the data set
  - TIMED = time in days, rounded to the nearest hundredth
  - SEX_c = recode SEX column from 0/1 to M/F
  - DOSE_f = factor of doses in ascending order
- Drop all columns between TAD and ADDL (inclusive), and DTTM
- Move TIMED next to the TIME column

Challenge: Create a new data frame using a pivot_* function to create a three-column data set, where the first column is the ID, the second column is the name of a covariate, and the third column is the value of a covariate

Problem #3: Plotting our data

Using the dataset from the previous problem, create the following plots:

- Plot 1: boxplot showing the distribution of age for each sex in your data set
  - hint: consider the number of rows per subject

- Plot 2: spaghetti plot (e.g. a line for each subject and a dot for each observation) of all subjects with each dose group uniquely colored, faceted by study day (DAY column), with the y-axis presented on a log scale
  - hint: use Factors to get doses to show up in order

Challenge: use summarize() and create a boxplot of individual Cmax values for each dose level

Problem #4: Writing a custom summary function

- Import mad_nonmem.csv
- Write a function that summarizes continuous covariates and outputs them in the following format: "`mean [5th percentile, 95th percentile]`"
- Apply the function separately for each dose group across the following covariates
  - WT, AGE, EGFR, BILI
- Pivot the data set so you have three columns: Dose, Covariate Name, and Covariate Summary Value

Challenge: Use `map()` to create a list of individual concentration-time plots

Backups

if_else() vs case_when()


```
race <- data.frame( id=1:5,
    racen=c(0:3,6)
)

race %>%
      mutate(
            racec = if_else(racen==0,"Caucasian",
            if_else(racen==1,"Black",
            if_else(racen==2,"Asian",
            if_else(racen==3,"Hispanic","missing"))))
      ))
```

id	racen	racec
1	0	Caucasian
2	1	Black
3	2	Asian
4	3	Hispanic
5	NA	missing

if_else() vs case_when()


```
race %>%
mutate(
   racec =
       case_when(
           racen == 0 ~ "Caucasian",
           racen == 1 ~ "Black",
           racen == 2 ~ "Asian",
           racen == 3 ~ "Hispanic",
           racen == 4 ~ "Other",
           TRUE ~ "missing"
       )
```

id	racen	racec
1	0	Caucasian
2	1	Black
3	2	Asian
4	3	Hispanic
5	NA	missing

Character Manipulation

- For working with strings we can use the stringr package, whose functions always start with str_* followed by a verb and the first argument

Function	Description
str_replace()	replace patterns
str_c()	combine characters
str_detect()	detect patterns
str_extract()	extract patterns
str_sub()	extract by position
str_length()	length of string

SEMI JOIN

idtime/idwt => in both

ID	TIME
1	0
1	1
2	0
2	1
3	0
3	1

idtime

ID	WT
1	70
2	80
4	75

idwt

semi_join(idtime, idwt)

ID	TIME
1	0
1	1
2	0
2	1

semi_join(idwt, idtime)

ID	WT
1	70
2	80

* A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x

```
dosing_df <- data.frame(ID = 1:2, TIME =0,
                        AMT = 100, MDV = 1)
sample_df <- data.frame(expand.grid(ID = 1:2, TIME = seq(0, 2, 1),
                                    AMT = 0, MDV = 0)

df3 <- bind_rows(sample_df,
                 dosing_df)
```

ID	TIME	AMT	MDV
1	0	0	0
2	0	0	0
1	1	0	0
2	1	0	0
1	2	0	0
2	2	0	0
1	0	100	1
2	0	100	1

* expand.grid is a very handy function for generating permutations
* MDV = missing dependent variable (a NONMEM-style flag column)

```
df3 %>% arrange(ID, TIME, desc(MDV))
```

ID	TIME	AMT	MDV
1	0	0	0
2	0	0	0
1	1	0	0
2	1	0	0
1	2	0	0
2	2	0	0
1	0	100	1
2	0	100	1

ID	TIME	AMT	MDV
1	0	100	1
1	0	0	0
1	1	0	0
1	2	0	0
2	0	100	1
2	0	0	0
2	1	0	0
2	2	0	0

Lecture 1: Base R Syntax

Introduction

- The R programming language was created by Ross Ihaka and Robert Gentleman at the University of Auckland in 1993
- R is based on the S programming language, a commercial product initially developed by Bell Labs
- R is free and open source
  - Open source = source code is available for modification and redistribution

How do we interact with R?

- Many users interact with R using the RStudio IDE (integrated development environment)

- Two main flavors of “programs” that you can write in R
  - R scripts
    - Basic script
  - Markdown scripts (Rmd or qmd)
    - Imposes structure on script
    - Produces customizable output documents in various formats (e.g. html, pdf)

- For either type, R will run the commands in the order that they are input, ignoring any line that begins with the comment (#) symbol

Basic Concepts

- Working Directory
  - The default location where R will look for files to load and where it will put any files you save
  - `> setwd("C:\home\directory")` – set working directory
  - `> getwd()` – get working directory
  - RStudio: files -> more -> set as working directory

- Projects in Rstudio
  - An RStudio Project keeps all of your R scripts, markdown documents, and data together in one place
  - Each project has its own directory, workspace, history, and source documents
    - Different analyses are kept separate from each other
  - RStudio: File -> New Project

Creating and using variables

- Use the “assign operator” <- to store the value from right hand side in a named variable on the left hand side
- Unlike some programming languages, in R you do not need to specify the data type for a new variable
- Variables can be overwritten, even as a different data type

```
test <- “hello world”
test
class(test)
test <- 11
test
class(test)
```

Functions

- An object which contains a series of instructions to perform a specific task
  - A function name is always followed by parentheses

```
log(x)
c(1, 1, 2, 3, 5)
```

- "Arguments" are "passed" into a function to determine the output
  - Some arguments are required, others are optional
  - Some arguments will have a default value
- Base R comes with many functions defined
- Install packages to load additional functions for specific tasks
- `help(log)` or `?mean` will open documentation in RStudio
- Users can easily define their own functions (covered in Lecture 4 today)

Data Types

- Numeric data are numbers that contain a decimal. Actually they can also be whole numbers but we'll gloss over that.
- Integers are whole numbers (those numbers without a decimal point).
- Logical data take on the value of either TRUE or FALSE. There's also another special type of logical called NA to represent missing values.
- Character data are used to represent string values. You can think of character strings as something like a word (or multiple words).
- Factor data are a special type of character string, with additional attributes (like levels and an order).

Data Types

- class() function will tell you what type an object is

```
x <- 12
class(x)
## [1] "numeric"
```

- Logical test to check object class

```
is.numeric(x)
## [1] TRUE
is.character(x)
## [1] FALSE
```

Typecasting

- Sometimes we may want to convert from one data type to another, this is called “typecasting”
- If R is unable to interpret the data in the requested format, it will return `NA`
  - `NA` is how R represents missing values
  - `NA`s are “contagious”, meaning that most operations on an `NA` will return an `NA`

```
as.character(1)
> “1”
as.numeric(“12.0”)
> 12.0
as.numeric(“A”)
> NA
```

Data Structures – Scalars and Vectors

- Scalar – an object containing a single value

```
x <- 12
```

- Vector – an object containing a set of elements, all of the same class

```
y <- 0:12
> y
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12

names <- c("Bob","Ralph","Mickey")
>names[2]
 [1] "Ralph"
```

scalar vector

Creating numeric vectors

```
1:5
[1] 1 2 3 4 5
seq(from=1, to=5, by=1)
[1] 1 2 3 4 5
rep(1,5)
[1] 1 1 1 1 1
c(4, 2, 3, 1, 5)
[1] 4 2 3 1 5
sort( c(4,2,3,1,5) )
[1] 1 2 3 4 5
```

NOTE: outputs from one function can be used as input to another function using nesting

Creating Factors

- Recall that Factors are an enumerated character data type
- We can think of these as labels corresponding to a numeric value

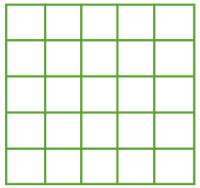
```
> factor(x=c(1,0,1,1,0), levels = 0:1, labels = c("Female", "Male"))
[1] Male   Female Male   Male   Female
Levels: Female Male
```

Data Structures – Matrices and Arrays

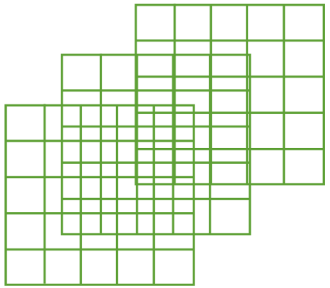
- Matrix – an object containing a square matrix of values, all of the same class

```
> mymat <- matrix(1:16, nrow = 4, byrow = TRUE)
> my_mat
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

- Array – a multi-dimensional array



matrix



array

Data Structures - Lists

- Lists are objects which contain other objects of various data types.

```
> mylist  <- list(colours = c("black", "yellow", "orange"),
                  evaluation = c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
                  time = matrix(1:6, nrow = 2))
> mylist
$colours
[1] "black"  "yellow" "orange"

$evaluation
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE

$time
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

MAJOR BENEFIT:
items in the list do not need to be the same length

Data Structures – Data Frames

- A data frame is a two-dimensional object made up of rows and columns
- Each column within the data frame can be of a different type
- All items must be the same length

```
> mydata <- data.frame(COLOR=c("Black","Yellow","Orange"),
                       EVALUATION=c(TRUE,TRUE,FALSE),
                       VALUE=c(1,3,6))
> mydata
   COLOR EVALUATION VALUE
1  Black       TRUE     1
2 Yellow       TRUE     3
3 Orange      FALSE     6
```

Data Structures – Data Frames

- Dimension of data frame

```
> dim(mydata)
[1] 3 3
```

- Accessing individual columns

```
> mydata$COLOR
[1] Black  Yellow Orange
Levels: Black Orange Yellow
> mydata[["VALUE"]]
[1] 1 3 6
```

- Accessing individual rows

```
> mydata[2,]
   COLOR EVALUATION VALUE
2 Yellow       TRUE     3
```

Some useful mathematical operations

`+ - * /`	arithmetic operations
`x^2`	x squared
`sqrt(x)`	square root of x
`y %% x`	remainder of y/x (modulo operator)
`round(x, digits=3)`	rounds x to the nearest thousandth
`floor(x)`	round down to nearest integer
`ceiling(x)`	round up to nearest integer
`sum(x)`	sum of all values in the vector x
`quantile(x, 0.9)`	90th percentile of the vector x
`mean(x)`	mean of the vector x

functions will operate on either scalar or every item in a vector with identical syntax

Mike's Unoffical Style Guide

- Avoid using more than 80 characters per line to allow reading the complete code
- Always use a space after a comma, never before
- The operators (==, +, -, <-,%>%, etc.) should have a space before and after
- There is no space between the name of a function and the first parenthesis, nor between the last argument and the final parenthesis of a function
- Avoid reusing names of functions and common variables (c <- 5 vs. c())
- Sort the script separating the parts with the comment form

```
# Import data -----
```

- Avoid accent marks or special symbols in names, files, routes, etc
- Object names should follow a constant structure (day_one vs. day_1)