

These materials adapted by Amelia McNamara from the RStudio CC BY-SA materials Introduction to R (2014) and Master the Tidyverse (2017).

Introduction to R & RStudio: deck 04: Syntax

Amelia McNamara

Visiting Assistant Professor of Statistical and Data Sciences
Smith College

January 2018

Syntax is the set of rules that govern what code works and doesn't work in a programming language. Most programming languages offer one standardized syntax, but R allows package developers to specify their own syntax. As a result, there is a large variety of (equally valid) R syntaxes.

R Syntax Comparison :: CHEAT SHEET

Dollar sign syntax

```
goal(data$x, data$y)
```

SUMMARY STATISTICS:

one continuous variable:

```
mean(mtcars$mpg)
```

one categorical variable:

```
table(mtcars$cyl)
```

two categorical variables:

```
table(mtcars$cyl, mtcars$am)
```

one continuous, one categorical:

```
mean(mtcars$mpg[mtcars$cyl==4])
```

```
mean(mtcars$mpg[mtcars$cyl==6])
```

```
mean(mtcars$mpg[mtcars$cyl==8])
```

PLOTTING:

one continuous variable:

```
hist(mtcars$disp)
```

```
boxplot(mtcars$disp)
```

one categorical variable:

```
barplot(table(mtcars$cyl))
```

two continuous variables:

```
plot(mtcars$disp, mtcars$mpg)
```

two categorical variables:

```
mosaicplot(table(mtcars$am, mtcars$cyl))
```

one continuous, one categorical:

```
histogram(mtcars$disp[mtcars$cyl==4])
```

```
histogram(mtcars$disp[mtcars$cyl==6])
```

```
histogram(mtcars$disp[mtcars$cyl==8])
```

```
boxplot(mtcars$disp[mtcars$cyl==4])
```

```
boxplot(mtcars$disp[mtcars$cyl==6])
```

```
boxplot(mtcars$disp[mtcars$cyl==8])
```

WRANGLING:

subsetting:

```
mtcars[mtcars$mpg>30, ]
```

making a new variable:

```
mtcars$efficient
```

```
mtcars$efficient
```

Formula syntax

```
goal(y~x|z, data=data, group=w)
```

SUMMARY STATISTICS:

one continuous variable:

```
mosaic::mean(~mpg, data=mtcars)
```

one categorical variable:

```
mosaic::tally(~cyl, data=mtcars)
```

two categorical variables:

```
mosaic::tally(cyl~am, data=mtcars)
```

one continuous, one categorical:

```
mosaic::mean(mpg~cyl, data=mtcars)
```

PLOTTING:

one continuous variable:

```
lattice::histogram(~disp, data=mtcars)
```

```
lattice::bwplot(~disp, data=mtcars)
```

one categorical variable:

```
mosaic::bargraph(~cyl, data=mtcars)
```

two continuous variables:

```
lattice::xyplot(mpg~disp, data=mtcars)
```

two categorical variables:

```
mosaic::tally(cyl~am, data=mtcars)
```

```
histogram(mtcars$disp[mtcars$cyl==4], data=mtcars)
```

```
histogram(mtcars$disp[mtcars$cyl==6], data=mtcars)
```

```
boxplot(mtcars$disp[mtcars$cyl==4], data=mtcars)
```

```
boxplot(mtcars$disp[mtcars$cyl==6], data=mtcars)
```

WRANGLING:

subsetting:

```
mtcars[mtcars$mpg>30, ]
```

making a new variable:

```
mtcars$efficient
```

```
mtcars$efficient
```

Tidyverse syntax

SUMMARY STATISTICS:

one continuous variable:

```
mtcars %>% summarise(mean_mpg = mean(mpg))
```

one categorical variable:

```
mtcars %>% summarise(n_cyl = n())
```

two categorical variables:

```
mtcars %>% summarise(n_am_cyl = n())
```

one continuous, one categorical:

```
mtcars %>% summarise(mean_mpg_by_cyl = mean(mpg ~ cyl))
```

PLOTTING:

one continuous variable:

```
mtcars %>% ggplot(aes(x=mpg)) + geom_histogram()
```

one categorical variable:

```
mtcars %>% ggplot(aes(x=cyl)) + geom_bar()
```

two continuous variables:

```
mtcars %>% ggplot(aes(x=disp, y=mpg)) + geom_point()
```

two categorical variables:

```
mtcars %>% ggplot(aes(x=factor(cyl), y=mpg)) + geom_bar()
```

one continuous, one categorical:

```
mtcars %>% ggplot(aes(x=disp, y=mpg)) + facet_grid(.~cyl)
```

one continuous, one categorical:

```
mtcars %>% ggplot(aes(x=disp, y=mpg)) + facet_grid(.~cyl)
```

one continuous, one categorical:

```
mtcars %>% ggplot(aes(x=disp, y=mpg)) + facet_grid(.~cyl)
```

WRANGLING:

subsetting:

```
mtcars %>% dplyr::filter(mpg>30)
```

making a new variable:

```
mtcars <- mtcars %>%
```

```
dplyr::mutate(efficient = if_else(mpg>30, TRUE, FALSE))
```

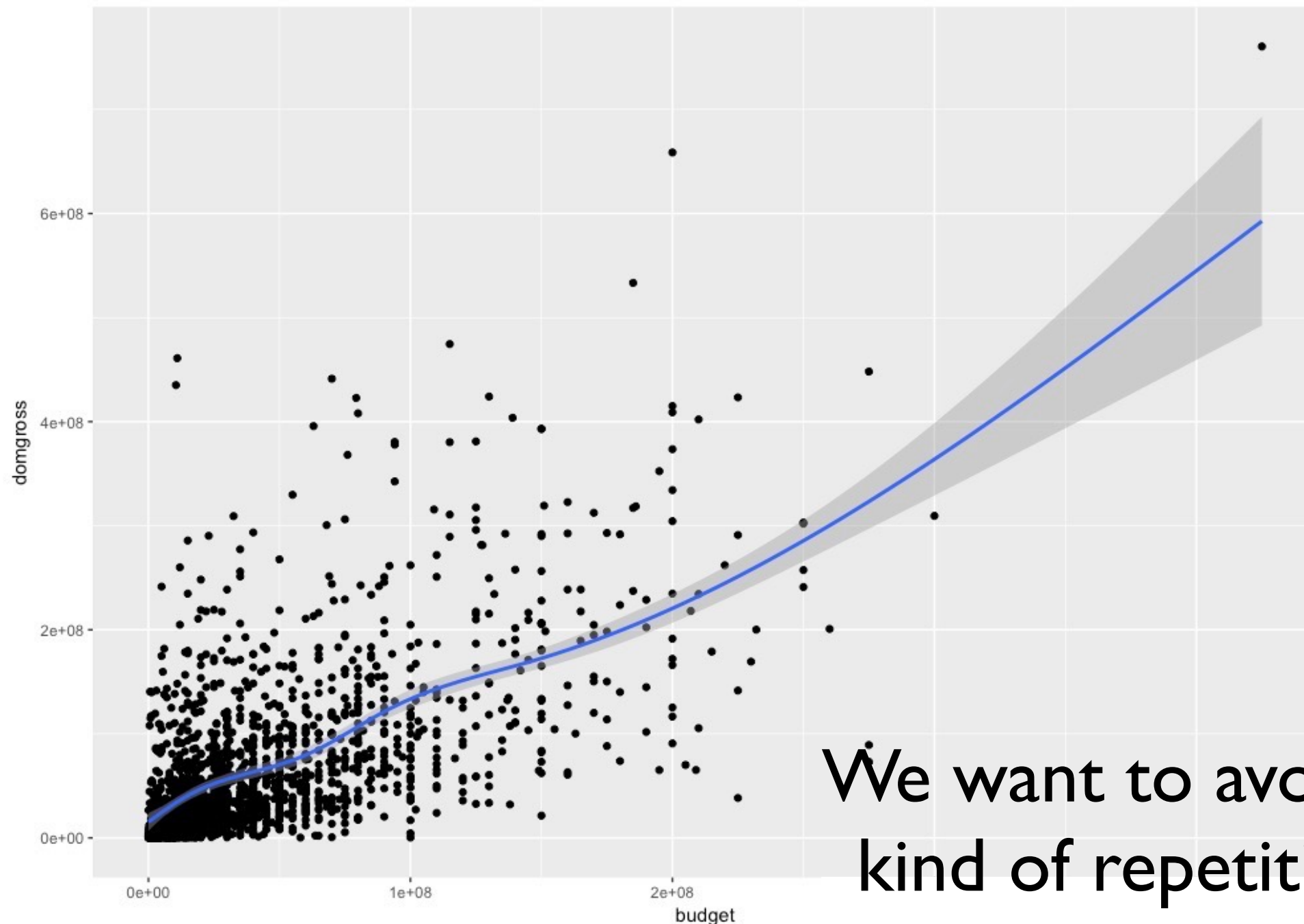
I've given you a copy of this cheatsheet in the "cheatsheets" folder of your workspace, and I have paper copies for anyone who wants one

the pipe

tilde

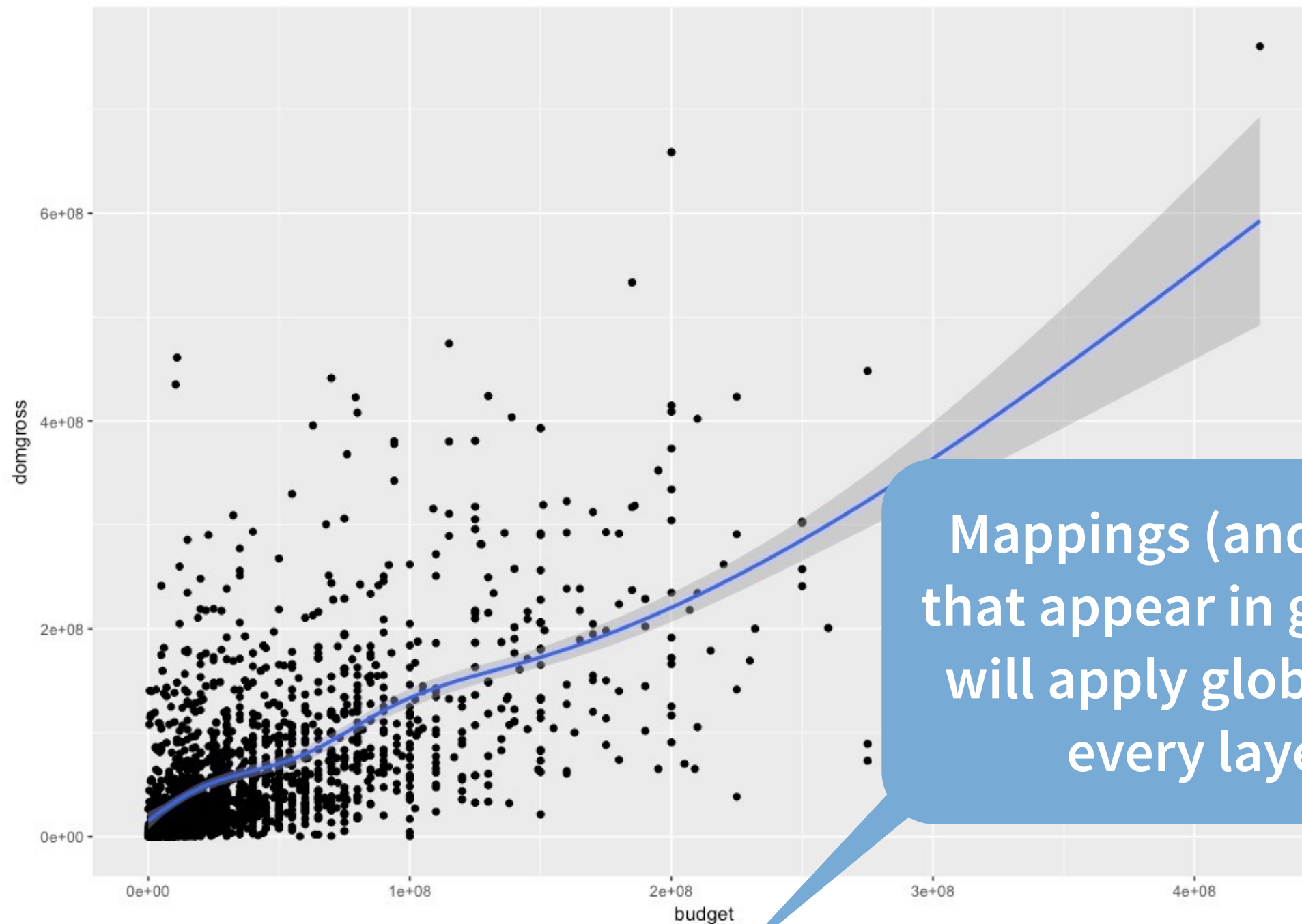
Read across the cheatsheet to see how different syntaxes approach the same problem

within
ggplot2



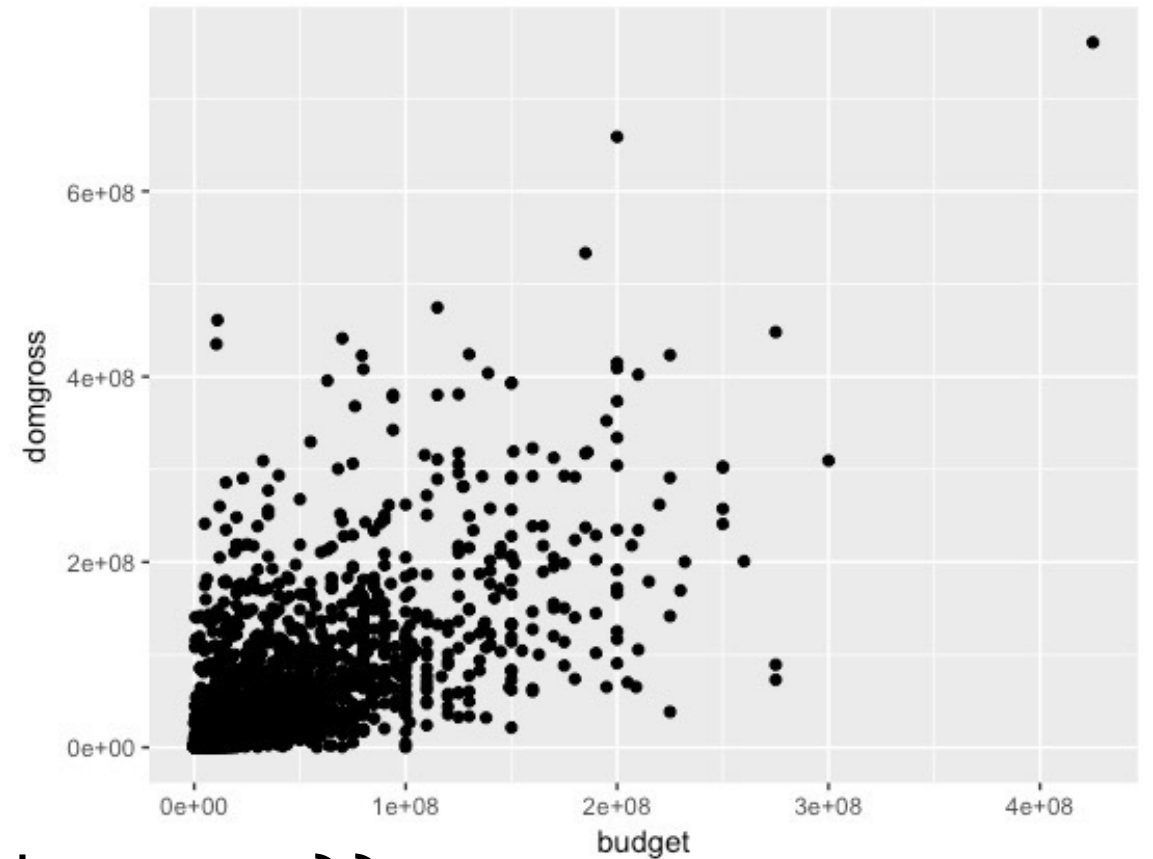
We want to avoid this
kind of repetition in
programming

```
ggplot(data = bechdel) +  
  geom_point(mapping = aes(x = budget, y = domgross)) +  
  geom_smooth(mapping = aes(x= budget, y = domgross))
```



```
ggplot(data = bechdel, aes(x= budget, y = domgross)) +  
  geom_point() +  
  geom_smooth()
```

MANY ways to say the same thing



```
ggplot(bechdel) +
```

```
  geom_point(aes(x = budget, y = domgross))
```

```
ggplot(bechdel, aes(x = budget, y = domgross)) +
```

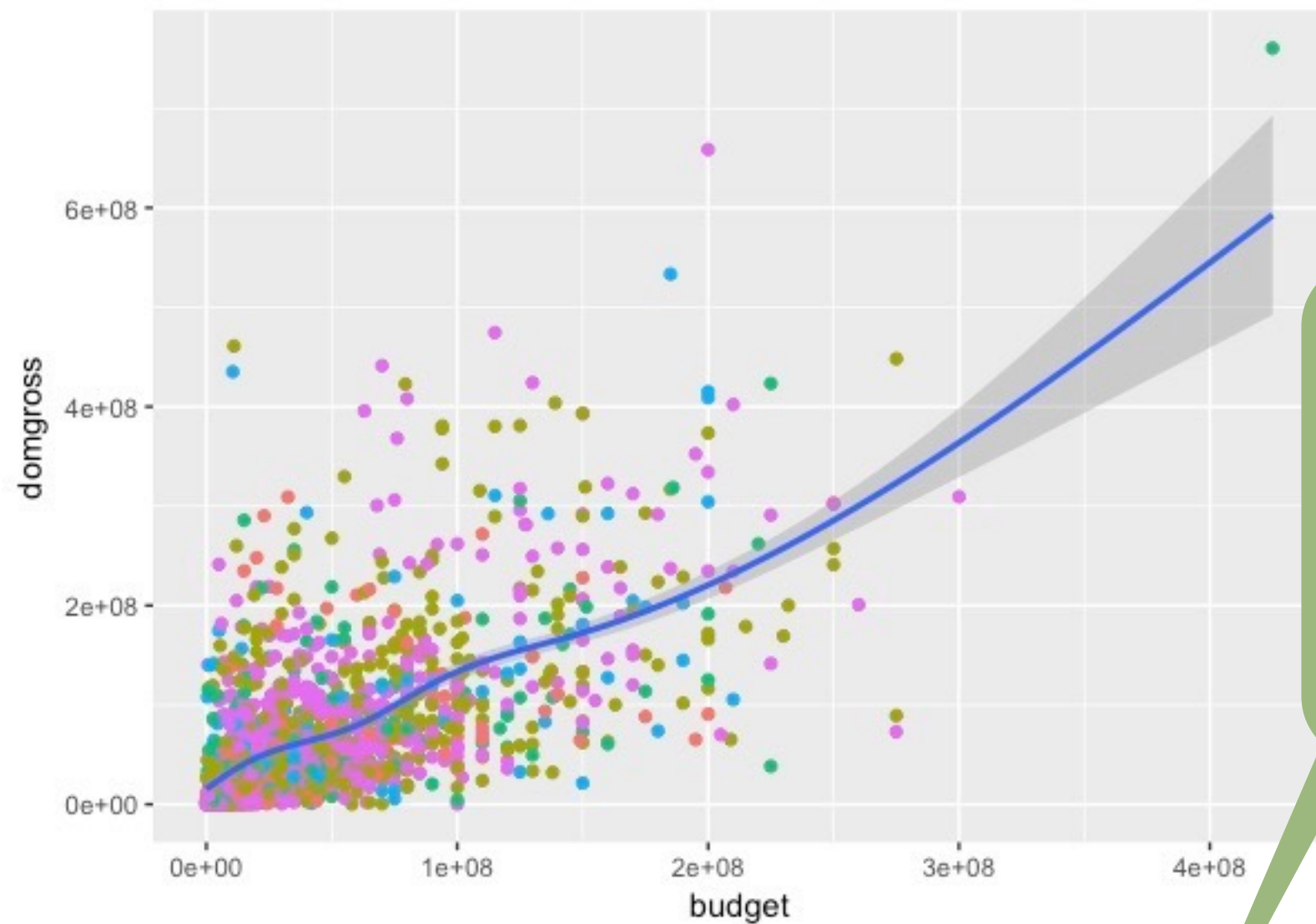
```
  geom_point()
```

```
ggplot(bechdel, aes(x= budget)) +
```

```
  geom_point(aes(y = domgross))
```

```
ggplot() +
```

```
  geom_point(bechdel, aes(x = budget, y = domgross))
```

Mappings (and data) that appear in a `geom_` function will add to or override the global mappings for that layer

```
ggplot(data = bechdel, aes(x= budget, y = domgross)) +  
  geom_point(aes(color=clean_test)) +  
  geom_smooth()
```

Subsetting

Toy data

```
beatles <- data.frame(  
  name = c("John", "Paul", "George", "Ringo"),  
  birth = c(1940, 1942, 1943, 1940),  
  instrument = c("guitar", "bass", "guitar", "drums")  
)
```

First—the
tidyverse way:
dplyr

dplyr methods for isolating data

`select()` - extract **variables**

`filter()` - extract **cases**

`arrange()` - reorder **cases**



select()

Extract columns by name.

```
select(.data, ...)
```

**data frame
to
transform**

**name(s) of columns to
extract**
(or a select helper function)

select()

Extract columns by name.

```
select(beatles, name, birth)
```

| name | birth | instrument | | name | birth |
|--------|-------|------------|---|--------|-------|
| John | 1940 | guitar | → | John | 1940 |
| Paul | 1942 | base | | Paul | 1942 |
| George | 1943 | guitar | | George | 1943 |
| Ringo | 1940 | drums | | Ringo | 1940 |

Your Turn 1

Alter the code to select just the **instrument** column:

```
select(beatles, name, birth)
```

01:00

select() helpers

: - Select range of columns

```
select(storms, storm:pressure)
```

- - Select every column but

```
select(storms, -c(storm, pressure))
```

starts_with() - Select columns that start with...

```
select(storms, starts_with("w"))
```

ends_with() - Select columns that end with...

```
select(storms, ends_with("e"))
```

select() helpers

contains() - Select columns whose names contain...

```
select(storms, contains("d"))
```

matches() - Select columns whose names match regular expression

```
select(storms, matches("^.{4}$"))
```

one_of() - Select columns whose names are one of a set

```
select(storms, one_of(c("storm", "storms", "Storm")))
```

num_range() - Select columns named in prefix, number style

```
select(storms, num_range("x", 1:5))
```



select() helpers

Data Transformation with dplyr Cheat Sheet

Studio

dplyr functions work with pipes and expect tidy data. In tidy data:

- Each variable is in its own column
- Each observation, or case, is in its own row

pipes
x %>% f(y)
becomes f(x, y)

Summarise Cases

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

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

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

Variations

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

Group Cases

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

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

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

ungroup(x, ...)
Returns ungrouped copy of table.
ungroup(g_iris)

Manipulate Cases

Extract Cases

Row functions return a subset of rows as a new table. Use a variant that ends in **_()** for non-standard evaluation friendly code.

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

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

sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, env = parent.frame())
Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, env = parent.frame())
Randomly select size rows.
sample_n(iris, 10, replace = TRUE)

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

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

Logical and boolean operators to use with filter()

| | | | | | |
|---|----|---------|------|---|-------|
| < | <= | is.na() | %in% | | xor() |
| > | >= | is.na() | ! | & | |

See **?base::logic** and **?Comparison** for help.

Arrange Cases

arrange(data, ...)
Order rows by values of a column (low to high). Use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

Add Cases

add_row(data, ..., before = NULL, after = NULL)
Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Manipulate Variables

Extract Variables

Column functions return a set of columns as a new table. Use a variant that ends in **_()** for non-standard evaluation friendly code.

select(data, ...)
Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

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

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

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

Make New Variables

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

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

transmute(data, ...)
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

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

mutate_at(tbl, cols, funs, ...)
Apply funs to specific columns. Use with **funs()** and the helper functions for **select()**.
mutate_at(iris, Species, funs(log(), log2(), log()))

mutate_if(tbl, predicate, funs, ...)
Apply funs to all columns of one type. Use with **funs()**.
mutate_if(iris, is.numeric, funs(log(), log2(), log()))

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

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

RStudio® is a trademark of RStudio, Inc. • © 2018 RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

Learn more with `browseVignettes(package = "dplyr", "tidyverse")` • dplyr 0.8.0 • tidyr 1.2.0 • Updated: 11/16

Extract Variables

Column functions return a set of columns as a new table. Use a variant that ends in **_()** for non-standard evaluation friendly code.



select(.data, ...)

Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

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

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

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

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



Now, the base R
way: brackets
and dollar signs

Base R bracket subset notation

in base R, you use the same syntax to

extract **variables**

extract **cases**

name of
object to
subset

brackets
(brackets always
mean subset)

vec[]

Subset notation

name of
object to
subset

vec

Subset notation

name of
object to
subset

brackets
(brackets always
mean subset)

`vec[?]`

an index
(that tells R which
elements to include)

Each dimension needs its own index!

vec[?]

| | | | | | |
|---|---|---|---|----|---|
| 6 | 1 | 3 | 6 | 10 | 5 |
|---|---|---|---|----|---|

Each dimension needs its own index!

vec[?]

| | | | | | |
|---|---|---|---|----|---|
| 6 | 1 | 3 | 6 | 10 | 5 |
|---|---|---|---|----|---|

Each dimension needs its own index!

`vec[?]`
`beatles[?, ?]`

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

Each dimension needs its own index!

`vec[?]`
`beatles[?, ?]`

which
rows to
include

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

Each dimension needs its own index!

`vec[?]`

`beatles[?, ?]`

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

which
rows to
include

which
columns
to include

Each dimension needs its own index!

`vec[?]`

`beatles[?, ?]`

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

which
rows to
include

separate
dimensions
with a
comma

which
columns
to include

Each dimension needs its own index!

`vec[?]`
`beatles[?, ?]`

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

What goes in the indexes?

Four ways to subset

1. Integers

2. Blank spaces

3. Names

4. Logical vectors (TRUE and FALSE)

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra


beatles[?, ?]

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

beatles[2, ?]

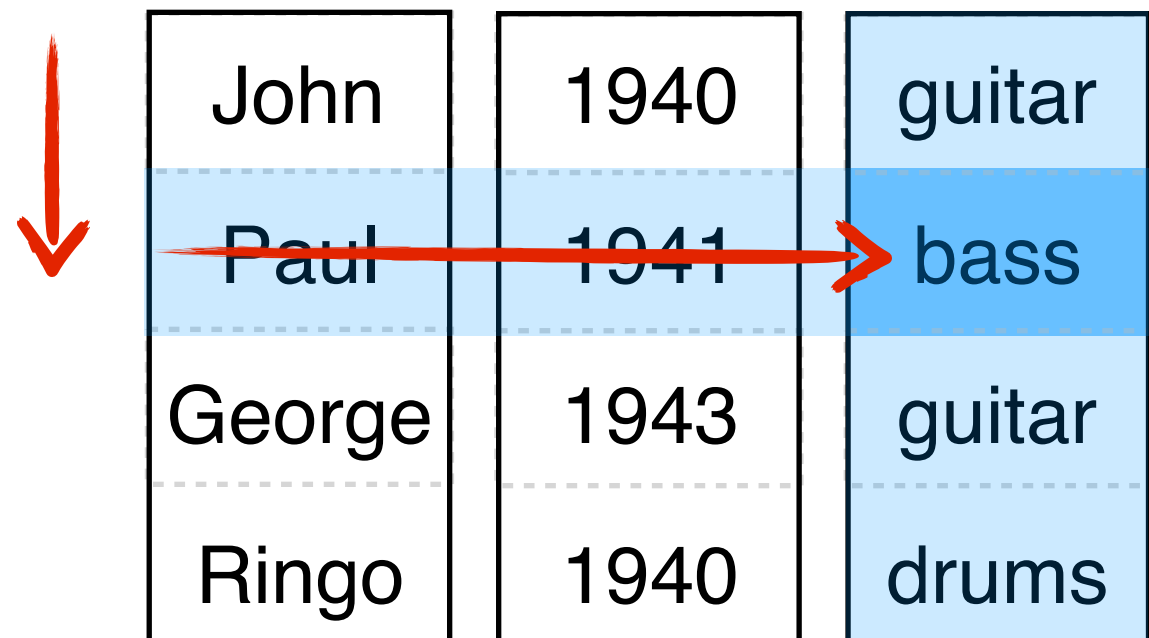


| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

beatles[2,3]



| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

beatles[2,3]

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

```
c("John","Paul",  
"George","Ringo")
```

```
c(1940, 1942,  
1943, 1940)
```

```
c("guitar",  
"drums")
```

John

1940

Paul

1942

George

guitar

Ringo

drums

beatles[2]

Names

If your object has names, you can ask for elements or columns back by name.

```
beatles[ , "birth"]
```

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

Names

If your object has names, you can ask for elements or columns back by name.

```
beatles[,c("name","birth")]
```

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

\$

The most common syntax for subsetting lists
and data frames

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

beatles\$birth

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

`beatles$birth`

name of
data frame

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

`beatles$birth`

name of
data frame

\$

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

beatles\$birth

name of
data frame

\$

name of column
(no quotes)



| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

`c(1940, 1941, 1943, 1940)`

`beatles$birth`

name of
data frame

\$

name of column
(no quotes)

Back to the
tidyverse

dplyr methods for isolating data

`select()` - extract **variables**

`filter()` - extract **cases**

`arrange()` - reorder **cases**



filter()

Extract rows that meet logical criteria.

```
filter(.data, ... )
```

**data frame to
transform**

**one or more logical
tests** (filter returns each
row for which the test is
TRUE)

common syntax

Each function takes a data frame / tibble as its first argument and returns a data frame / tibble.

```
filter(.data, ... )
```

dplyr function

**data frame to
transform**

**function
specific
arguments**

filter()

Extract rows that meet logical criteria.

```
filter(beatles, name == "George")
```

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1942 | base |
| George | 1943 | guitar |
| Ringo | 1940 | drums |



| | | |
|--------|------|--------|
| George | 1943 | guitar |
|--------|------|--------|

filter()

Extract rows that meet logical criteria.

```
filter(beatles, name == "George")
```

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1942 | base |
| George | 1943 | guitar |
| Ringo | 1940 | drums |

= sets
(returns nothing)

== tests if equal
(returns TRUE or FALSE)

Logical comparisons

Logical comparisons

?Comparison

| | |
|------------------------|--------------------------|
| <code>x < y</code> | Less than |
| <code>x > y</code> | Greater than |
| <code>x == y</code> | Equal to |
| <code>x <= y</code> | Less than or equal to |
| <code>x >= y</code> | Greater than or equal to |
| <code>x != y</code> | Not equal to |
| <code>x %in% y</code> | Group membership |
| <code>is.na(x)</code> | Is NA |
| <code>!is.na(x)</code> | Is not NA |

Logical comparisons

What will these return?

`1 < 3`

`1 > 3`

`c(1, 2, 3, 4, 5) > 3`

%in%

What does this do?

```
1 %in% c(1, 2, 3, 4)
```

```
1 %in% c(2, 3, 4)
```

```
c(3,4,5,6) %in% c(2, 3, 4)
```

%in%

%in% tests whether the object on the left is a member of the group on the right.

```
1 %in% c(1, 2, 3, 4)
```

```
# TRUE
```

```
1 %in% c(2, 3, 4)
```

```
# FALSE
```

```
c(3,4,5,6) %in% c(2, 3, 4)
```

```
# TRUE TRUE FALSE FALSE
```

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | | |
| <code>x < 3</code> | | |
| <code>x <= 3</code> | | |
| <code>x == 3</code> | | |
| <code>x != 3</code> | | |
| <code>x = 3</code> | | |

01:00

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | <code>c(F, F, T, T, T)</code> | greater than or equal to |
| <code>x < 3</code> | | |
| <code>x <= 3</code> | | |
| <code>x == 3</code> | | |
| <code>x != 3</code> | | |
| <code>x = 3</code> | | |

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | <code>c(F, F, T, T, T)</code> | greater than or equal to |
| <code>x < 3</code> | <code>c(T, T, F, F, F)</code> | less than |
| <code>x <= 3</code> | | |
| <code>x == 3</code> | | |
| <code>x != 3</code> | | |
| <code>x = 3</code> | | |

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | <code>c(F, F, T, T, T)</code> | greater than or equal to |
| <code>x < 3</code> | <code>c(T, T, F, F, F)</code> | less than |
| <code>x <= 3</code> | <code>c(T, T, T, F, F)</code> | less than or equal to |
| <code>x == 3</code> | | |
| <code>x != 3</code> | | |
| <code>x = 3</code> | | |

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | <code>c(F, F, T, T, T)</code> | greater than or equal to |
| <code>x < 3</code> | <code>c(T, T, F, F, F)</code> | less than |
| <code>x <= 3</code> | <code>c(T, T, T, F, F)</code> | less than or equal to |
| <code>x == 3</code> | <code>c(F, F, T, F, F)</code> | equal to |
| <code>x != 3</code> | | |
| <code>x = 3</code> | | |

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | <code>c(F, F, T, T, T)</code> | greater than or equal to |
| <code>x < 3</code> | <code>c(T, T, F, F, F)</code> | less than |
| <code>x <= 3</code> | <code>c(T, T, T, F, F)</code> | less than or equal to |
| <code>x == 3</code> | <code>c(F, F, T, F, F)</code> | equal to |
| <code>x != 3</code> | <code>c(T, T, F, T, T)</code> | not equal to |
| <code>x = 3</code> | | |

Your turn

`x <- c(1, 2, 3, 4, 5)`

| Operator | Result | Comparison |
|------------------------|-------------------------------|--------------------------|
| <code>x > 3</code> | <code>c(F, F, F, T, T)</code> | greater than |
| <code>x >= 3</code> | <code>c(F, F, T, T, T)</code> | greater than or equal to |
| <code>x < 3</code> | <code>c(T, T, F, F, F)</code> | less than |
| <code>x <= 3</code> | <code>c(T, T, T, F, F)</code> | less than or equal to |
| <code>x == 3</code> | <code>c(F, F, T, F, F)</code> | equal to |
| <code>x != 3</code> | <code>c(T, T, F, T, T)</code> | not equal to |
| <code>x = 3</code> | | same as <- |

Your Turn 2

Alter the code to filter out just rows where birth is 1940:

```
filter(beatles, name == "George")
```

01:00

filter()

Extract rows that meet *every* logical criteria.

```
filter(beatles, birth==1940, instrument == "guitar")
```

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1942 | base |
| George | 1943 | guitar |
| Ringo | 1940 | drums |



| | | |
|------|------|--------|
| John | 1940 | guitar |
|------|------|--------|

Boolean operators

Boolean operators

?base::Logic

| | |
|------------------------|------------|
| <code>a & b</code> | and |
| <code>a b</code> | or |
| <code>xor(a,b)</code> | exactly or |
| <code>!a</code> | not |

Boolean operators

You can combine logical tests with `&`, `|`, `xor`, `!`, `any`, and `all`

$$x > 2 \ \& \ x < 9$$

Boolean operators

You can combine logical tests with &, |, xor, !, any, and all

$x > 2 \ \& \ x < 9$



TRUE &

Boolean operators

You can combine logical tests with &, |, xor, !, any, and all

$x > 2 \ \& \ x < 9$



TRUE & TRUE

Boolean operators

You can combine logical tests with &, |, xor, !, any, and all

$x > 2 \ \& \ x < 9$



TRUE & TRUE



TRUE

&

Are both condition 1 **and** condition 2 true?

| expression | outcome |
|---------------|---------|
| TRUE & TRUE | TRUE |
| TRUE & FALSE | FALSE |
| FALSE & TRUE | FALSE |
| FALSE & FALSE | FALSE |

|

Is either condition 1 **or** condition 2 true?

| expression | outcome |
|---------------|---------|
| TRUE TRUE | TRUE |
| TRUE FALSE | TRUE |
| FALSE TRUE | TRUE |
| FALSE FALSE | FALSE |

xor

Is either condition 1 **or** condition 2 true, **but not both**?

| expression | outcome |
|--------------------|---------|
| xor(TRUE, TRUE) | FALSE |
| xor(TRUE, FALSE) | TRUE |
| xor(FALSE, TRUE) | TRUE |
| xor(FALSE, FALSE) | FALSE |

!

Negation

| expression | outcome |
|------------|---------|
| !(TRUE) | FALSE |
| !(FALSE) | TRUE |

filter()

Extract rows that meet *every* logical criteria.

```
filter(beatles, birth==1940 & instrument == "guitar")
```

| name | birth | instrument |
|--------|-------|------------|
| John | 1940 | guitar |
| Paul | 1942 | base |
| George | 1943 | guitar |
| Ringo | 1940 | drums |



| | | |
|------|------|--------|
| John | 1940 | guitar |
|------|------|--------|

Base R

Logical

You can subset with a logical vector of the same length as the dimension you are subsetting. Each element that corresponds to a TRUE will be returned.

```
beatles[c(FALSE,TRUE,TRUE,FALSE), ]
```

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |



Logical

You can subset with a logical vector of the same length as the dimension you are subsetting. Each element that corresponds to a TRUE will be returned.

```
beatles[c(FALSE,TRUE,TRUE,FALSE), ]
```

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |



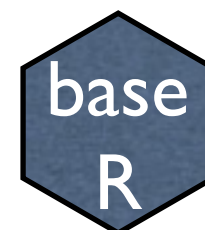
Logical

You can provide a statement that **evaluates** to a logical to get something similar to a `dplyr filter()` statement.

```
beatles[beatles$birth == 1940, ]
```

```
beatles[c(TRUE, FALSE, FALSE, TRUE), ]
```

| | | |
|--------|------|--------|
| John | 1940 | guitar |
| Paul | 1941 | bass |
| George | 1943 | guitar |
| Ringo | 1940 | drums |



Logical tests

?Comparison

| | |
|------------------------|--------------------------|
| <code>x < y</code> | Less than |
| <code>x > y</code> | Greater than |
| <code>x == y</code> | Equal to |
| <code>x <= y</code> | Less than or equal to |
| <code>x >= y</code> | Greater than or equal to |
| <code>x != y</code> | Not equal to |
| <code>x %in% y</code> | Group membership |
| <code>is.na(x)</code> | Is NA |
| <code>!is.na(x)</code> | Is not NA |

Boolean operators

?base::Logic

| | |
|------------------------|------------|
| <code>a & b</code> | and |
| <code>a b</code> | or |
| <code>xor(a,b)</code> | exactly or |
| <code>!a</code> | not |

Bigger example

Baby names

We'll play with the babynames dataset in R. It comes in its own package, so you will need to load the package (what command does that?) and then View the data

| | year | sex | name | n | prop |
|----|------|-----|-----------|------|-------------|
| 1 | 1880 | F | Mary | 7065 | 0.072384329 |
| 2 | 1880 | F | Anna | 2604 | 0.026679234 |
| 3 | 1880 | F | Emma | 2003 | 0.020521700 |
| 4 | 1880 | F | Elizabeth | 1939 | 0.019865989 |
| 5 | 1880 | F | Minnie | 1746 | 0.017888611 |
| 6 | 1880 | F | Margaret | 1578 | 0.016167370 |
| 7 | 1880 | F | Ida | 1472 | 0.015081349 |
| 8 | 1880 | F | Alice | 1414 | 0.014487111 |
| 9 | 1880 | F | Bertha | 1320 | 0.013524036 |
| 10 | 1880 | F | Sarah | 1288 | 0.013196180 |
| 11 | 1880 | F | Annie | 1258 | 0.012888816 |
| 12 | 1880 | F | Clara | 1226 | 0.012560961 |
| 13 | 1880 | F | Ella | 1156 | 0.011843777 |
| 14 | 1880 | F | Florence | 1063 | 0.010890947 |
| 15 | 1880 | F | Cor | 1045 | 0.010706528 |

Your Turn 3

See if you can use filter and/or select to get just the babies with your first name. (If you have a name with less than 5 occurrences in the US in any year, pick a neighbor's name.)

01:00

Your Turn 4

See if you can use the logical operators to show:

- All of the names where **prop** is greater than or equal to 0.08
- All of the children named “Sea”
- All of the names that have a missing value for **n**
(Hint: this should return an empty data set).

04:00


```
filter(babynames, prop >= 0.08)
```

| # | year | sex | name | n | prop |
|-----|------|-----|---------|------|------------|
| # 1 | 1880 | M | John | 9655 | 0.08154630 |
| # 2 | 1880 | M | William | 9531 | 0.08049899 |
| # 3 | 1881 | M | John | 8769 | 0.08098299 |

```
filter(babynames, name == "Sea")
```

| # | year | sex | name | n | prop |
|-----|------|-----|------|---|--------------|
| # 1 | 1982 | F | Sea | 5 | 2.756771e-06 |
| # 2 | 1985 | M | Sea | 6 | 3.119547e-06 |
| # 3 | 1986 | M | Sea | 5 | 2.603512e-06 |
| # 4 | 1998 | F | Sea | 5 | 2.580377e-06 |

```
filter(babynames, is.na(n))
```

```
# 0 rows
```

Two common mistakes

1. Using `=` instead of `==`

```
filter(babynames, name = "Sea")  
filter(babynames, name == "Sea")
```

2. Forgetting quotes

```
filter(babynames, name == Sea)  
filter(babynames, name == "Sea")
```


Your Turn 5

Use Boolean operators to alter the code below to return only the rows that contain:

- Girls named Sea
- Names that were used by exactly 5 or 6 children in 1880
- Names that are one of Acura, Lexus, or Yugo

```
filter(babynames, name == "Sea" | name == "Anemone")
```

04:00


```
filter(babynames, name == "Sea", sex == "F")
```

| # | year | sex | name | n | prop |
|-----|------|-----|------|---|--------------|
| # 1 | 1982 | F | Sea | 5 | 2.756771e-06 |
| # 2 | 1998 | F | Sea | 5 | 2.580377e-06 |

```
filter(babynames, n == 5 | n == 6, year == 1880)
```

| # | year | sex | name | n | prop |
|-------|------|-----|--------|-----|--------------|
| # 1 | 1880 | F | Abby | 6 | 6.147289e-05 |
| # 2 | 1880 | F | Aileen | 6 | 6.147289e-05 |
| # ... | ... | ... | ... | ... | ... |

```
filter(babynames, name %in% c("Acura", "Lexus", "Yugo"))
```

| # | year | sex | name | n | prop |
|-------|------|-----|-------|-----|--------------|
| # 1 | 1990 | F | Lexus | 36 | 1.752932e-05 |
| # 2 | 1990 | M | Lexus | 12 | 5.579156e-06 |
| # ... | ... | ... | ... | ... | ... |

Two more common mistakes

3. Collapsing multiple tests into one

```
filter(babynames, 10 < n < 20)  
filter(babynames, 10 < n, n < 20)
```

4. Stringing together many tests (when you could use %in%)

```
filter(babynames, n == 5 | n == 6 | n == 7 | n == 8)  
filter(babynames, n %in% c(5, 6, 7, 8))
```

Saving
results

Saving results

Print to screen

```
filter(babynames, name %in% c("Acura", "Lexus", "Yugo"))
```

| # | year | sex | name | n | prop |
|-------|------|-----|-------|-----|--------------|
| # 1 | 1990 | F | Lexus | 36 | 1.752932e-05 |
| # 2 | 1990 | M | Lexus | 12 | 5.579156e-06 |
| # ... | ... | ... | ... | ... | ... |

Save to new data frame (where does this appear?)

```
carnames <- filter(babynames, name %in% c("Acura", "Lexus", "Yugo"))
```

Save over existing data frame (**dangerous!**)

```
babynames <- filter(babynames, name %in% c("Acura", "Lexus", "Yugo"))
```

```
babynames <- filter(babynames, name %in% c("Acura", "Lexus", "Yugo"))
```

Uh oh...

```
rm(babynames)  
str(babynames)
```

Phew!

Your Turn 5

Try to:

- Filter out the babynames data for your name. (You may want to **also** filter for your gender)
- Assign that data its own name
- Use your subsetting data to create a ggplot graphic of the popularity of your name over time


```
amelia <- babynames %>%  
  filter(name=="Amelia" & sex == "F")
```

```
ggplot(amelia) + geom_point(aes(x=year, y=n))
```

```
ggplot(amelia) + geom_line(aes(x=year, y=n))
```

