STAT 209
The Grammar of Data-Wrangling

June 24, 2021

Colin Reimer Dawson

Outline

- The Pipe, Clarified
- Data Wrangling
- "Verbs" for manipulating single data frames

Outline

What's the Deal with %>%?

- Provided by `magrittr` package (but is loaded by `dplyr` which is loaded by `tidyverse`)
- Semantics based on UNIX command line "pipe" operator (|)

What's the Deal with %>%?

- Provided by `magrittr` package (but is loaded by `dplyr` which is loaded by `tidyverse`)
- Semantics based on UNIX command line "pipe" operator (|)

- The work "The Treachery of Images" (1929) by Rene Magritte

What's the Deal with %>%?

- Provided by `magrittr` package (but is loaded by `dplyr` which is loaded by `tidyverse`)
- Semantics based on UNIX command line "pipe" operator (|)

- The work "The Treachery of Images" (1929) by Rene Magritte

Neither one is an actual pipe

- When we add or multiply three numbers, we usually write it like

```
2 + 5 + 8
```

```
[1] 15
```

- When we add or multiply three numbers, we usually write it like

```
2 + 5 + 8
```

```
     [1] 15
```

- But when we calculate the result, we (probably) do it left to right. Something like:

```
firstTwo <- 2+5
firstTwo + 8
```

```
     [1] 15
```

- When we add or multiply three numbers, we usually write it like

```
2 + 5 + 8
```

```
     [1] 15
```

- But when we calculate the result, we (probably) do it left to right. Something like:

```
firstTwo <- 2+5
firstTwo + 8
```

```
     [1] 15
```

- In essence we're applying the + "function", first to 2 and 5, then to the result of that and 8

- When we add or multiply three numbers, we usually write it like

```
2 + 5 + 8
```

```
     [1] 15
```

- But when we calculate the result, we (probably) do it left to right. Something like:

```
firstTwo <- 2+5
firstTwo + 8
```

```
     [1] 15
```

- In essence we're applying the + "function", first to 2 and 5, then to the result of that and 8
- We could have written this as

```
firstTwo <- `+`(2,5)
`+`(firstTwo, 8)
```

```
     [1] 15
```

- When we add or multiply three numbers, we usually write it like

```
2 + 5 + 8
```

```
     [1] 15
```

- But when we calculate the result, we (probably) do it left to right. Something like:

```
firstTwo <- 2+5
firstTwo + 8
```

```
     [1] 15
```

- In essence we're applying the + "function", first to 2 and 5, then to the result of that and 8
- We could have written this as

```
firstTwo <- `+`(2,5)
`+`(firstTwo, 8)
```

```
     [1] 15
```

- Or even

```
`+`(`+`(2,5), 8)
```

```
     [1] 15
```

The Pipe is a Syntactic Convenience

- The %>% operator lets us "unroll" multi-step operations by passing the output of each one to the next one

```
library(Stat2Data)
data(Pulse)
```

This:

```
Pulse %>% head(n = 3)
```

```
  Active Rest Smoke Sex Exercise Hgt Wgt
1     97   78     0   1        1  63 119
2     82   68     1   0        3  70 225
3     88   62     0   0        3  72 175
```

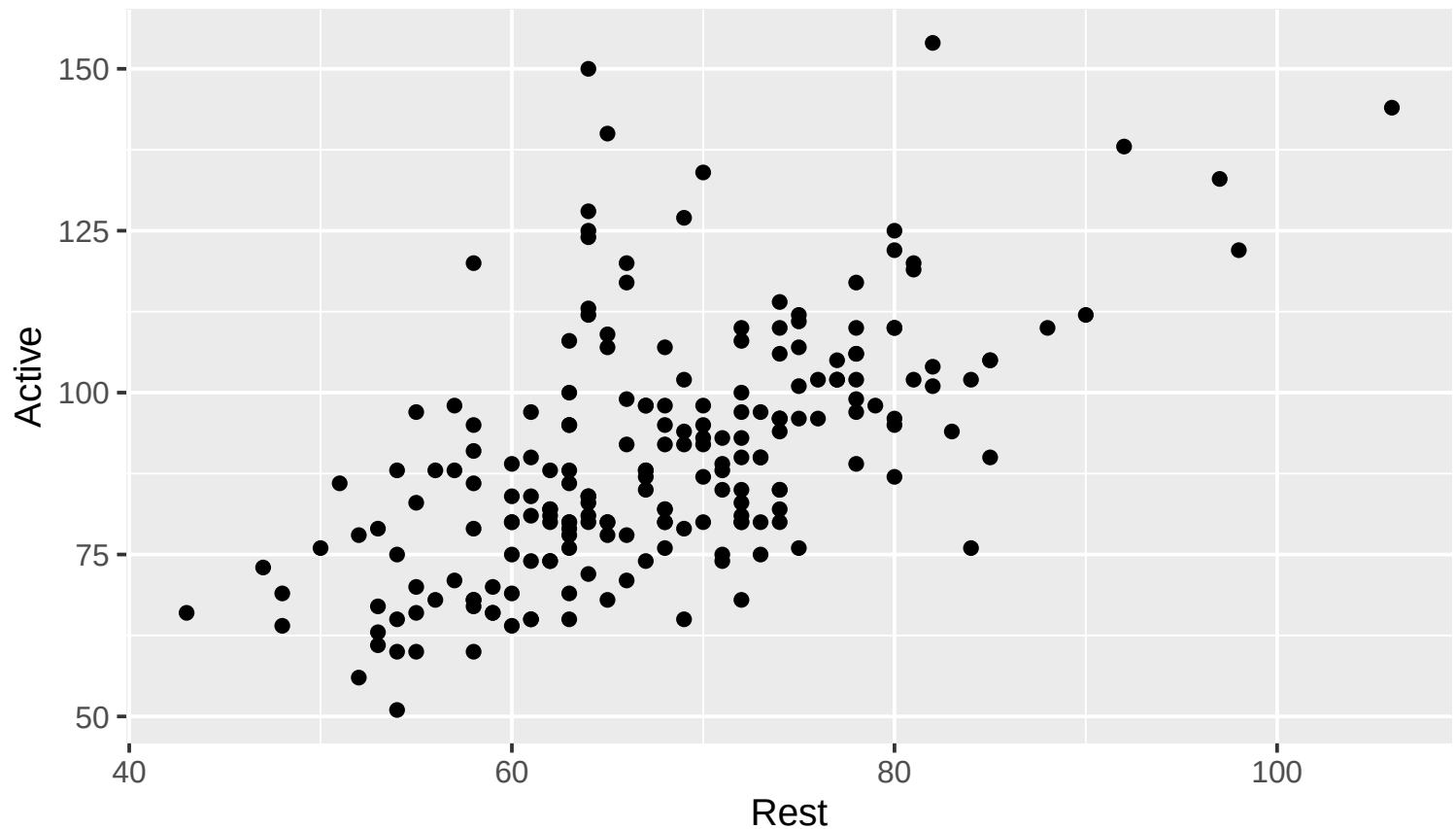
is equivalent to

```
head(Pulse, n = 3)
```

```
  Active Rest Smoke Sex Exercise Hgt Wgt
1     97   78     0   1        1  63 119
2     82   68     1   0        3  70 225
3     88   62     0   0        3  72 175
```

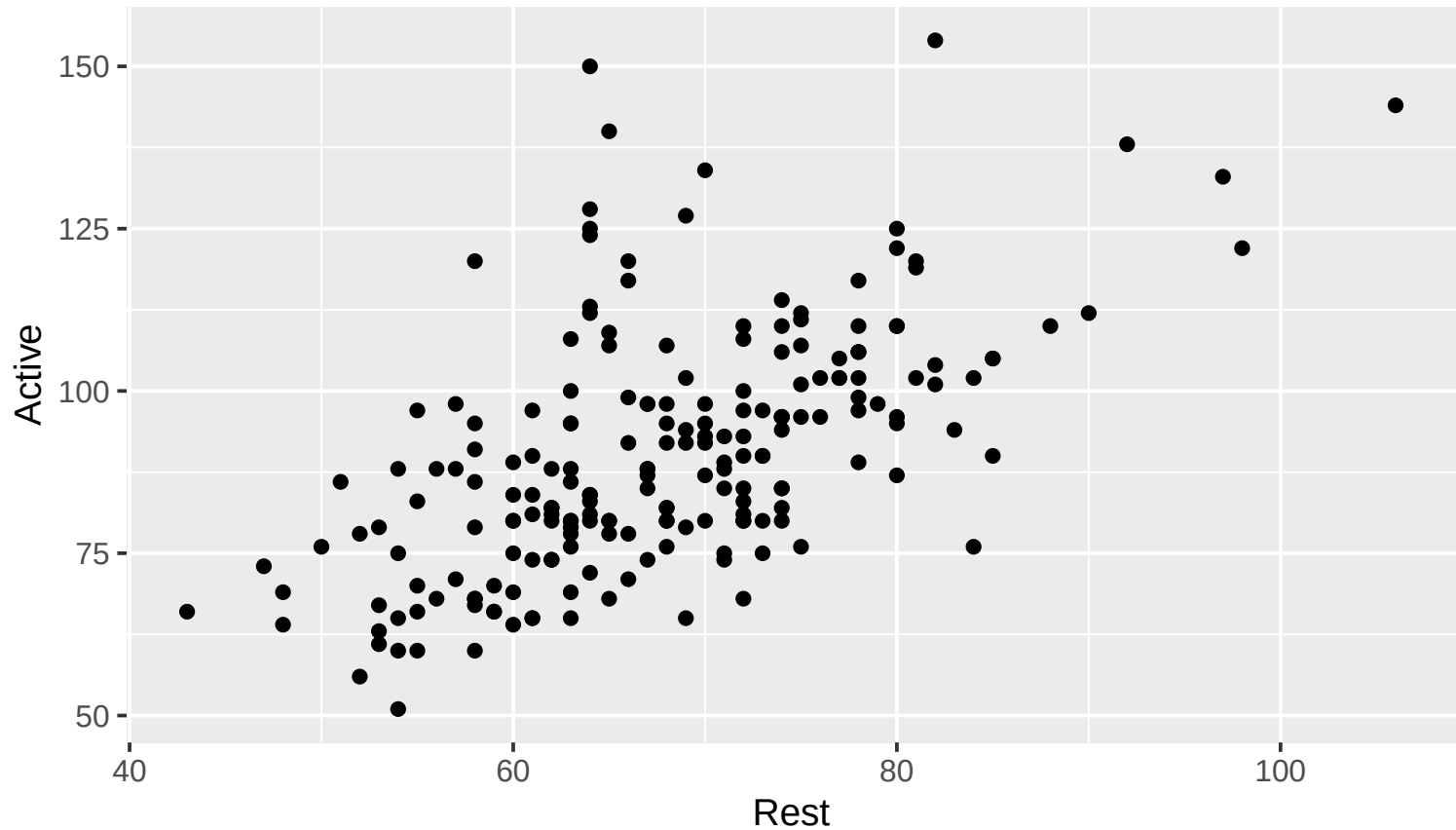
And this

```
Pulse %>%
    filter(Smoke == 0) %>%
    ggplot(aes(x = Rest, y = Active)) +
    geom_point()
```



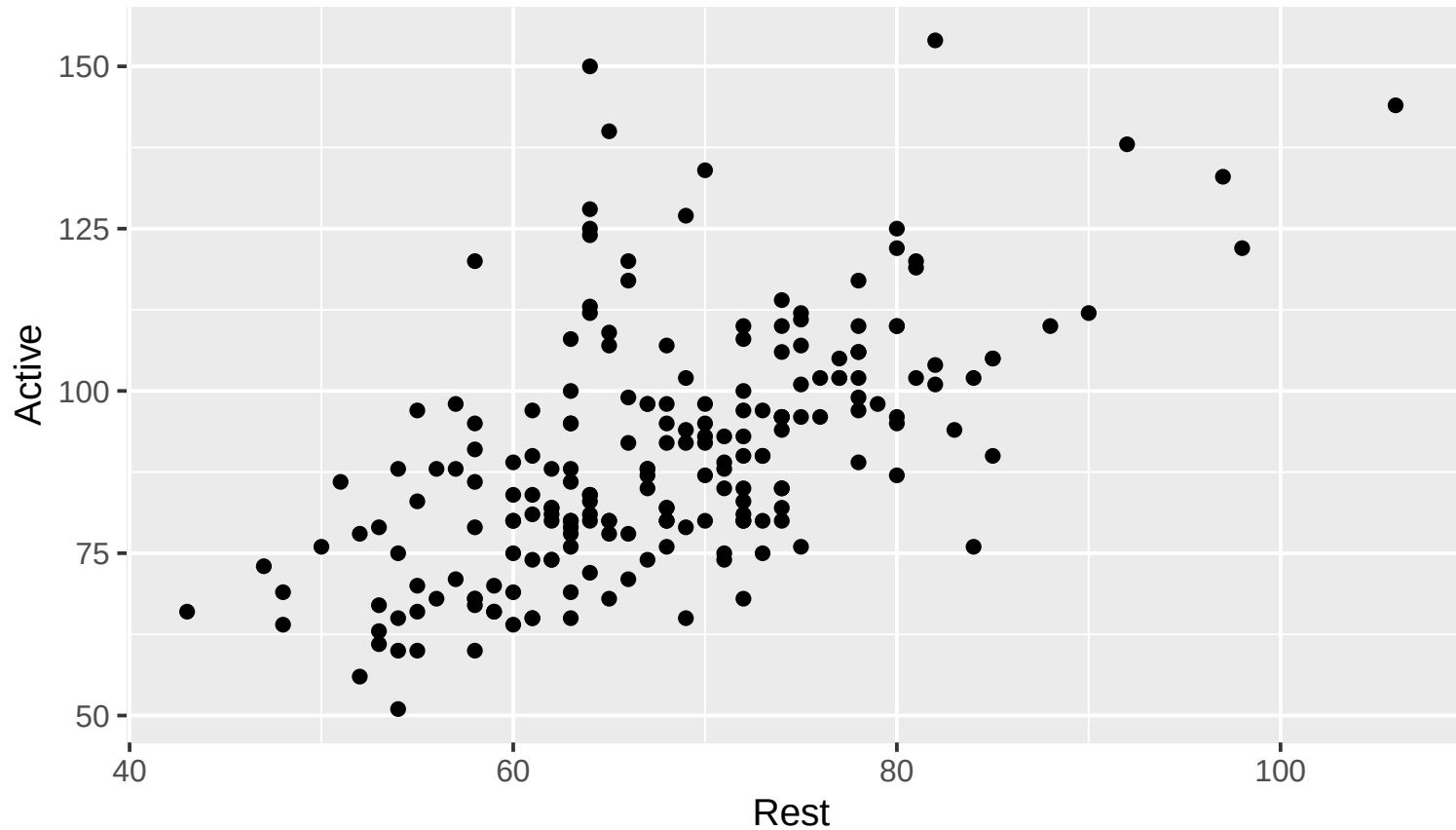
is equivalent to

```
ggplot(filter(Pulse, Smoke == 0), aes(x = Rest, y = Active)) +
    geom_point()
```



is equivalent to

```
ggplot(filter(Pulse, Smoke == 0), aes(x = Rest, y = Active)) +
    geom_point()
```



... but which is easier to read?

Little Bunny Foo Foo

Little bunny foo foo
Hopping through the forest
Scooping up the field mice
and bopping them on the head

Little Bunny Foo Foo

Little bunny foo foo
Hopping through the forest
Scooping up the field mice
and bopping them on the head

Or, as (fake) R code:

```
foo_foo <- little_bunny()
bop(scoop(hop(foo_foo, place = forest), target = field_mice), bodypart = head)
```

Exercise: Rewrite the second line using the pipe operator

Outline

What is Data-Wrangling?

- Often (more often than not?) our data doesn't come to us in "visualization-ready form"
- Before we can examine the patterns and relationships we care about, need to take "wild" data and **"wrangle"** it into a useable form

The `dplyr` package

- Part of the `tidyverse`
- One of the cleanest and most popular approaches to **reproducible data wrangling**
- Organized around data-wrangling **verbs** (functions), based on database ideas in SQL

Outline

Five Data-Wrangling Verbs

`filter()`	Extract a **subset of cases** (rows) that meet some **condition**
`select()`	Extract a **subset of variables** (columns) either by name or according to a condition
`mutate()`	Create **new variables** that apply **case by case**
`arrange()`	**Sort cases** on some criteria
`summarize()`	Compute some **summary statistic(s)** for variable(s), **combining cases**

Other Useful Verbs

`group_by()`	Split the data into **subsets** according to a **categorical variable** (often used together with `summarize()`)
`rename()`	Replace **variable names**
`*_join()`	A family of functions for **merging** datasets (later)
`do()`	Perform a task **repeatedly** (later)

All of these are summarized with minimal examples on the reference sheet in RStudio Help or (here)

Data Transformation with dplyr : : CHEAT SHEET

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))

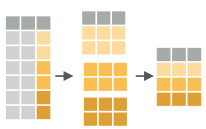
count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
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))

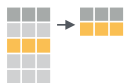
group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

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

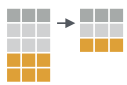
Manipulate Cases

EXTRACT CASES

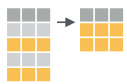
Row functions return a subset of rows as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.



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

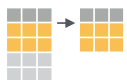


distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values. Also **distinct_()**.
distinct(iris, Species)



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)*

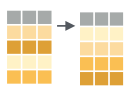
top_n(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

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

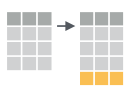
See **?base::logic** and **?Comparison** for help.

ARRANGE CASES



arrange(.data, …) Order rows by values of a column or columns (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)

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

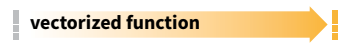
select(.data, …)
Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

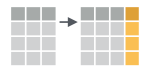
Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match)	**num_range**(prefix, range)	**:**, e.g. mpg:cyl
ends_with(match)	**one_of**(…)	**-**, e.g, -Species
matches(match)	**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).





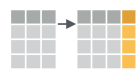
mutate(.data, …)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

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(.)))

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

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

The General Approach

- Five main verbs take in a **dataset** and **return a modified dataset**

The General Approach

- Five main verbs take in a **dataset** and **return a modified dataset**
- We can chain them via the **pipe** (%>%) to perform **multiple operations in sequence**

The General Approach

- Five main verbs take in a **dataset** and **return a modified dataset**
- We can chain them via the **pipe** (%>%) to perform **multiple operations in sequence**
- Get to know individual verbs well so you can combine them in creative/efficient/readable ways

Example: Biometric Data

```
library(Stat2Data)
data(Pulse) # data is included with the Stat2Data package
## head() returns the first n cases
Pulse %>%
    head(n = 10)
```

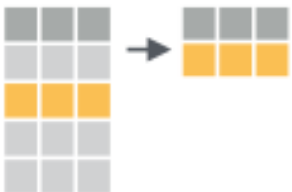
	Active	Rest	Smoke	Sex	Exercise	Hgt	Wgt
1	97	78	0	1	1	63	119
2	82	68	1	0	3	70	225
3	88	62	0	0	3	72	175
4	106	74	0	0	3	72	170
5	78	63	0	1	3	67	125
6	109	65	0	0	3	74	188
7	66	43	0	1	3	67	140
8	68	65	0	0	3	70	200
9	100	63	0	0	1	70	165
10	70	59	0	1	2	65	115

`filter()`: Select cases meeting a criterion

From the reference sheet:

EXTRACT CASES

Row functions return a subset of rows as a new table.



filter(.data, …**)** Extract rows that meet logical criteria.
filter(mtcars, mpg > 20)

Example: Extract Non-Smokers

```
library(tidyverse)
## Note: head() is just for display purposes
Pulse %>%
    filter(Smoke == 0) %>%
    head(n = 5)
```

```
  Active Rest Smoke Sex Exercise Hgt Wgt
1     97   78     0   1        1  63 119
2     88   62     0   0        3  72 175
3    106   74     0   0        3  72 170
4     78   63     0   1        3  67 125
5    109   65     0   0        3  74 188
```

Example: Extract a Random Subset

```
Pulse %>%
    sample_n(size = 5)
```

```
  Active Rest Smoke Sex Exercise Hgt Wgt
1     97   72     0   1        1  64 135
2     88   57     0   0        3  76 230
3    125   80     1   1        3  65 125
4     69   63     0   0        2  69 140
5     65   54     0   0        3  74 190
```

`select()`: Extract particular columns

select(.data, …**)** Extract columns as a table. Also **select_if()**.
select(mtcars, mpg, wt)

Use these helpers with select() and across()
e.g. select(mtcars, mpg:cyl)

contains(match**)**	**num_range(**prefix, range**)**	**:**, e.g. mpg:cyl
ends_with(match**)**	**one_of(**…**)**	**-**, e.g, -gear
matches(match**)**	**starts_with(**match**)**	**everything()**

Example: Include Specific Variables

```
## head() just for display purposes
Pulse %>%
    select(Active, Rest, Hgt, Wgt) %>%
    head(n = 5)
```

```
  Active Rest Hgt Wgt
1     97   78  63 119
2     82   68  70 225
3     88   62  72 175
4    106   74  72 170
5     78   63  67 125
```

Example: Exclude Specific Variables

```
## head() just for display purposes
Pulse %>%
    select(-Smoke, -Sex, -Exercise) %>%
    head(n = 5)
```

```
  Active Rest Hgt Wgt
1     97   78  63 119
2     82   68  70 225
3     88   62  72 175
4    106   74  72 170
5     78   63  67 125
```

Example: Select Contiguous Columns

```
## head() just for display purposes
Pulse %>%
    select(Smoke:Exercise) %>%
    head(n = 5)
```

```
  Smoke Sex Exercise
1     0   1        1
2     1   0        3
3     0   0        3
4     0   0        3
5     0   1        3
```

Example: Variables With a Suffix

```
## This is very silly in this case
Pulse %>%
    select(ends_with("e")) %>%
    head(n = 5)
```

```
  Active Smoke Exercise
1     97     0        1
2     82     1        3
3     88     0        3
4    106     0        3
5     78     0        3
```

mutate(): Define New Variables

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, …, .before = NULL, .after = NULL**)**
Compute new column(s). Also **add_column(), add_count(),** and **add_tally().**
mutate(mtcars, gpm = 1/mpg)

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

Example: Convert Pulse from BPM to MPB

```
Pulse %>%
    mutate(
        InvActive = 1/Active,
        InvRest   = 1/Rest) %>%
    head(n = 5)
```

	Active	Rest	Smoke	Sex	Exercise	Hgt	Wgt	InvActive	InvRest
1	97	78	0	1	1	63	119	0.010309278	0.01282051
2	82	68	1	0	3	70	225	0.012195122	0.01470588
3	88	62	0	0	3	72	175	0.011363636	0.01612903
4	106	74	0	0	3	72	170	0.009433962	0.01351351
5	78	63	0	1	3	67	125	0.012820513	0.01587302

Example: Convert and Drop Originals

```
Pulse %>%
    transmute(
        InvActive = 1 / Active, InvRest = 1 / Rest,
        Male      = 1 - Sex,
        Wgt_kg    = Wgt / 2.2,
        Hgt_m     = Hgt / 39.37) %>%
    head(n = 5)
```

```
  InvActive    InvRest Male    Wgt_kg    Hgt_m
1 0.010309278 0.01282051    0  54.09091 1.600203
2 0.012195122 0.01470588    1 102.27273 1.778004
3 0.011363636 0.01612903    1  79.54545 1.828804
4 0.009433962 0.01351351    1  77.27273 1.828804
5 0.012820513 0.01587302    0  56.81818 1.701803
```

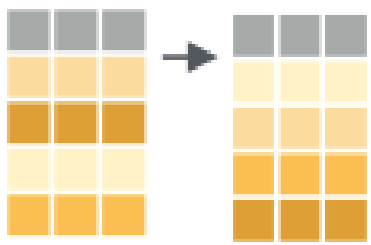
Example: Log transform several at once

```
## Note: mutate_at() transforms variables "in place", so
## be sure to relabel if needed
Pulse %>%
    mutate_at(vars(Active, Rest, Wgt, Hgt), list(log)) %>%
    head()
```

	Active	Rest	Smoke	Sex	Exercise	Hgt	Wgt
1	4.574711	4.356709	0	1	1	4.143135	4.779123
2	4.406719	4.219508	1	0	3	4.248495	5.416100
3	4.477337	4.127134	0	0	3	4.276666	5.164786
4	4.663439	4.304065	0	0	3	4.276666	5.135798
5	4.356709	4.143135	0	1	3	4.204693	4.828314
6	4.691348	4.174387	0	0	3	4.304065	5.236442

arrange(): Sort the rows based on a column

ARRANGE CASES



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

arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

Example: Sort by Resting HR

```
Pulse %>%
    arrange(Rest) %>%
    head(n = 5)
```

```
  Active Rest Smoke Sex Exercise Hgt Wgt
1     66   43     0   1        3  67 140
2     73   47     0   1        3  62 145
3     69   48     0   1        3  67 140
4     64   48     0   0        3  69 170
5     76   50     0   0        3  71 208
```

Example: Sort in Descending Order

```
Pulse %>%
    arrange(desc(Rest)) %>%
    head(n = 5)
```

```
  Active Rest Smoke Sex Exercise Hgt Wgt
1    144  106     0   1        1  62 140
2    122   98     0   1        1  62 105
3    133   97     0   1        1  65 130
4    110   95     1   0        1  69 180
5    121   94     1   0        1  74 250
```

summarize(): Compute statistics across rows

Summarise Cases

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

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

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

Example: Compute Some Means

```
Pulse %>%
    summarize( # Note: The American spelling works too
        n         = n(),
        AvgRest   = mean(Rest),
        AvgActive = mean(Active),
        AvgHeight = mean(Hgt),
        AvgWeight = mean(Wgt))
```

```
     n  AvgRest AvgActive AvgHeight AvgWeight
1 232 68.34914  91.29741  68.24569  157.9181
```

Example: Compute Some Means and Medians Compactly

```
Pulse %>%
    summarize_at(
        vars(Rest, Active),
        list(Mean = mean, Median = median))
```

```
  Rest_Mean Active_Mean Rest_Median Active_Median
1  68.34914    91.29741          68          88.5
```

Example: Count Smokers/Non-Smokers

```
## troubleshooting tip: the mosaic package uses some of the
## same names as dplyr, so it can cause conflicts
## leading to errors or unexpected output
if("mosaic" %in% (.packages())) detach(package:mosaic)
```

```
Pulse %>%
    count(Smoke)
```

```
  Smoke   n
1     0 206
2     1  26
```

Example: Summarize by Group

```
Pulse %>%
    group_by(Smoke) %>%
    summarize(
        n        = n(),
        MeanRest = mean(Rest),
        SDRest   = sd(Rest))


 # A tibble: 2 x 4
   Smoke     n MeanRest SDRest
   <int> <int>    <dbl>  <dbl>
 1     0   206     67.8   9.85
 2     1    26     72.8   9.80
```

Plot With Means and Confidence Intervals

```
PulseSummary <-
    Pulse %>%
    group_by(Smoke) %>%
    summarize(
        n    = n(),
        Mean = mean(Rest),
        SD   = sd(Rest))
```

Plot With Means and Confidence Intervals

```
smoker_heartrate_plot <- PulseSummary %>%
    ggplot(aes(x = factor(Smoke))) +
    scale_x_discrete(
        name   = "Smoking Status",
        breaks = c(0,1),
        labels = c("Non-smoker", "Smoker")) +
    ylab("Mean Resting Heart Rate (bpm)") +
    geom_point(aes(y = Mean)) +
    geom_errorbar(
        aes(
            ymin = Mean - 1.96 * SD / sqrt(n),
            ymax = Mean + 1.96 * SD / sqrt(n)),
        width = 0.1) # the default width is enormous
```

Plot With Means and Confidence Intervals

```
smoker_heartrate_plot
```