

R Workshop Featuring dplyr

Jorge Loría

October 20th, 2020

Where can we find this presentation?

<https://github.com/Jels95/Dplyr-Workshop>

What is dplyr?????

What is dplyr?????

Dplyr is a package that permits a *nice* interface in R to work with data.frames.

What is dplyr?????

Dplyr is a package that permits a *nice* interface in R to work with data.frames.

Ok, nice, but what is a data.frame?

¹categorical

²<https://vita.had.co.nz/papers/tidy-data.pdf>

What is dplyr?????

Dplyr is a package that permits a *nice* interface in R to work with data.frames.

Ok, nice, but what is a data.frame?

A data.frame is like a matrix of dimensions $n \times p$, where we have several different types of data. Each column corresponds to a single variable, and each variable has a specific type (numeric, string, logical, factor¹). Each row should correspond to a single observation².

¹categorical

²<https://vita.had.co.nz/papers/tidy-data.pdf>

What dataset are we going to use?

We are going to use a dataset from tidytuesday, about Himalayan Climbers

What dataset are we going to use?

We are going to use a dataset from tidytuesday, about Himalayan Climbers



Figure 1: Himalaya? It must be easy to survive there

How to access it?

```
install.packages('tidytuesdayR')  
library(tidytuesdayR)  
himalaya <- tidytuesdayR::tt_load('2020-09-22')  
members <- himalaya$members
```

Let's start with questions

What questions can we ask this dataset that you think would be interesting to know?
(Please write them in the chat!)

Let's start with questions

What questions can we ask this dataset that you think would be interesting to know?
(Please write them in the chat!)

The idea is to be able to solve most, if not all of those questions by the end of the workshop!

Let's start with questions

What questions can we ask this dataset that you think would be interesting to know?
(Please write them in the chat!)

The idea is to be able to solve most, if not all of those questions by the end of the workshop!

We will study some tools, and you should be able to answer them by the end of the workshop. If I can see that's not possible, I'll tell you what libraries are good to answer them. Ok?

Let's take a look:

```
members %>%  
  head()
```

```
## # A tibble: 6 x 21  
##   expedition_id member_id peak_id peak_name   year season sex      age ci  
##   <chr>          <chr>    <chr>   <chr>    <dbl> <chr>  <chr> <dbl> <chr>  
## 1 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      40 Fra  
## 2 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      41 Fra  
## 3 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      27 Fra  
## 4 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      40 Fra  
## 5 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      34 Fra  
## 6 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      25 Fra  
## # ... with 12 more variables: expedition_role <chr>, hired <lgl>,  
## #   highpoint_metres <dbl>, success <lgl>, solo <lgl>, oxygen_used <lgl>,  
## #   died <lgl>, death_cause <chr>, death_height_metres <dbl>, injured <lgl>,  
## #   injury_type <chr>, injury_height_metres <dbl>
```

%>%

This is a function that doesn't do much, but does a lot. It allows to compose functions (as the math people do) but in a way that permits an easy reading of the functions, and what is happening.

%>%

This is a function that doesn't do much, but does a lot. It allows to compose functions (as the math people do) but in a way that permits an easy reading of the functions, and what is happening. So, instead of writing: $f(g(h(i(x))))$

%>%

This is a function that doesn't do much, but does a lot. It allows to compose functions (as the math people do) but in a way that permits an easy reading of the functions, and what is happening. So, instead of writing: $f(g(h(i(x))))$, we write:

```
x %>%  
  i() %>%  
  h() %>%  
  g() %>%  
  f()
```

Which is arguably easier to read than the previous expression. Specially if some of those functions had extra arguments.

The Verbs

There are 6 main verbs in `dplyr` that we will study:

function	action
<code>filter</code>	keeps rows that satisfy a condition
<code>arrange</code>	sorts the rows following the order
<code>select</code>	keeps/eliminates the columns by name
<code>mutate</code>	creates new variables from existing variables
<code>summarise</code>	summarises the data
<code>group_by*</code>	groups under specific conditions

The Verbs

All the verbs work very similarly (in `dplyr`, in English verbs are more confusing):

The Verbs

All the verbs work very similarly (in `dplyr`, in English verbs are more confusing):

- ▶ first argument is a dataframe,

The Verbs

All the verbs work very similarly (in `dplyr`, in English verbs are more confusing):

- ▶ first argument is a dataframe,
- ▶ arguments describe what to do with the dataframe, using the existing variables (without quotations).

The Verbs

All the verbs work very similarly (in `dplyr`, in English verbs are more confusing):

- ▶ first argument is a dataframe,
- ▶ arguments describe what to do with the dataframe, using the existing variables (without quotations).
- ▶ output is a dataframe

The Verbs

All the verbs work very similarly (in `dplyr`, in English verbs are more confusing):

- ▶ first argument is a dataframe,
- ▶ arguments describe what to do with the dataframe, using the existing variables (without quotations).
- ▶ output is a dataframe

The Verbs

All the verbs work very similarly (in `dplyr`, in English verbs are more confusing):

- ▶ first argument is a dataframe,
- ▶ arguments describe what to do with the dataframe, using the existing variables (without quotations).
- ▶ output is a dataframe

This structure allows us to concatenate (`%>%`) simple operations to obtain complex results.

filter



Figure 2: Another type of filters

filter

This function removes rows that don't satisfy a (or several) condition that we specify. The arguments it receives are logical, and will use it to do that removal:

```
library(dplyr) ## load the library
members %>%
  filter(oxygen_used)
```

```
## # A tibble: 18,233 x 21
```

##		expedition_id	member_id	peak_id	peak_name	year	season	sex	age
##		<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>
##	1	ANN170101	ANN17010~	ANN1	Annapurn~	1970	Spring	M	35
##	2	ANN170102	ANN17010~	ANN1	Annapurn~	1970	Spring	M	28
##	3	ANN170102	ANN17010~	ANN1	Annapurn~	1970	Spring	M	32
##	4	ANN177301	ANN17730~	ANN1	Annapurn~	1977	Autumn	M	37
##	5	ANN177301	ANN17730~	ANN1	Annapurn~	1977	Autumn	M	29
##	6	ANN177301	ANN17730~	ANN1	Annapurn~	1977	Autumn	M	28
##	7	ANN178301	ANN17830~	ANN1	Annapurn~	1978	Autumn	F	35

filter

We can use several columns to filter, and can even modify them. Let's see what people older than 75 years **didn't** need to use oxygen

```
members %>%  
  filter(!oxygen_used, age > 75)
```

```
## # A tibble: 27 x 21
```

##	expedition_id	member_id	peak_id	peak_name	year	season	sex	age	
##	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>	
##	1	GIMM93301	GIMM9330~	GIMM	Gimmigel~	1993	Autumn	M	80
##	2	GIME93301	GIME9330~	GIME	Gimmigel~	1993	Autumn	M	80
##	3	PUM005104	PUM00510~	PUMO	Pumori	2005	Spring	M	76
##	4	AMAD06347	AMAD0634~	AMAD	Ama Dabl~	2006	Autumn	M	76
##	5	AMAD07314	AMAD0731~	AMAD	Ama Dabl~	2007	Autumn	M	77
##	6	YAKA08201	YAKA0820~	YAKA	Yakawa K~	2008	Summer	F	77
##	7	AMAD08331	AMAD0833~	AMAD	Ama Dabl~	2008	Autumn	M	78
##	8	TASH10302	TASH1030~	TASH	Tashi Ka~	2010	Autumn	M	76

filter

We can also use several columns at once to do a filter. Let's see what climber(s?) died a little bit after getting injured:

```
members %>%  
  filter(death_height_metres > injury_height_metres)
```

```
## # A tibble: 1 x 21  
##   expedition_id member_id peak_id peak_name   year season sex      age ci  
##   <chr>          <chr>    <chr>   <chr>    <dbl> <chr>  <chr> <dbl> <ch  
## 1 BAUD70101      BAUD7010~ BAUD     Baudha     1970 Spring M          26 Jap  
## # ... with 12 more variables: expedition_role <chr>, hired <lgl>,  
## #   highpoint_metres <dbl>, success <lgl>, solo <lgl>, oxygen_used <lgl>,  
## #   died <lgl>, death_cause <chr>, death_height_metres <dbl>, injured <lgl>,  
## #   injury_type <chr>, injury_height_metres <dbl>
```

Exercises filter

Exercises filter

- ▶ From those who were not injured, how many died in 1979?

Exercises filter

- ▶ From those who were not injured, how many died in 1979?
- ▶ How many of those that succeeded, were doing it solo and got injured above 1000 meters?

Exercises filter

- ▶ From those who were not injured, how many died in 1979?
- ▶ How many of those that succeeded, were doing it solo and got injured above 1000 meters?
- ▶ How many people died on their climb below 1000 meters?

Exercises filter

- ▶ From those who were not injured, how many died in 1979?
- ▶ How many of those that succeeded, were doing it solo and got injured above 1000 meters?
- ▶ How many people died on their climb below 1000 meters?

Exercises filter

- ▶ From those who were not injured, how many died in 1979?
- ▶ How many of those that succeeded, were doing it solo and got injured above 1000 meters?
- ▶ How many people died on their climb below 1000 meters?

filter saving

Let's now save the dataframe of the members that went on a expedition in 1905:

```
members1905 <- members %>%  
  filter(year == 1905)
```

arrange

arrange



Figure 3: Another type of arrangement

arrange

arrange

This verb sorts the data frame with the column(s) that we tell it to use

```
members %>%  
  arrange(year)
```

```
## # A tibble: 76,519 x 21
```

##	expedition_id	member_id	peak_id	peak_name	year	season	sex	age	
##	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>	
##	1	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	29
##	2	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA
##	3	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA
##	4	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA
##	5	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA
##	6	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA
##	7	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	36
##	8	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA
##	9	KANG05201	KANG0520~	KANG	Kangchen~	1905	Summer	M	NA

arrange on characters

Let's go back to the 1905 dataset, and check how it orders when we use a string, instead of a number:

```
members1905 %>%  
  arrange(member_id)
```

```
## # A tibble: 9 x 21
```

```
##   expedition_id member_id peak_id peak_name  year season sex    age ci  
##   <chr>          <chr>    <chr>  <chr>    <dbl> <chr> <chr> <dbl> <chr>  
## 1 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      29 UK  
## 2 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      36 Sw  
## 3 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Sw  
## 4 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Sw  
## 5 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Ita  
## 6 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Nep  
## 7 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Nep  
## 8 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Nep
```

arrange on strings

`arrange` uses the lexicographic order, to sort when it encounters letters/characters, so: $a < A < b < B \dots < z < Z$. If you're not sure if a number or a symbol is smaller than another one, you can try it out on the 'Console' in R for example type `'1' < 'a'`. Is this true? Or false?

arrange on strings

`arrange` uses the lexicographic order, to sort when it encounters letters/characters, so: $a < A < b < B \dots < z < Z$. If you're not sure if a number or a symbol is smaller than another one, you can try it out on the 'Console' in R for example type `'1' < 'a'`. Is this true? Or false? TRUE!

arrange on strings

`arrange` uses the lexicographic order, to sort when it encounters letters/characters, so: $a < A < b < B \dots < z < Z$. If you're not sure if a number or a symbol is smaller than another one, you can try it out on the 'Console' in R for example type `'1' < 'a'`. Is this true? Or false? TRUE! How about `'&' > '2'`?

arrange on strings

`arrange` uses the lexicographic order, to sort when it encounters letters/characters, so: $a < A < b < B \dots < z < Z$. If you're not sure if a number or a symbol is smaller than another one, you can try it out on the 'Console' in R for example type `'1' < 'a'`. Is this true? Or false? TRUE! How about `'&' > '2'`? FALSE!

arrange on strings

`arrange` uses the lexicographic order, to sort when it encounters letters/characters, so: $a < A < b < B \dots < z < Z$. If you're not sure if a number or a symbol is smaller than another one, you can try it out on the 'Console' in R for example type `'1' < 'a'`. Is this true? Or false? TRUE! How about `'&' > '2'`? FALSE! Is `' ' < '1'` (space less than one)?

arrange on strings

`arrange` uses the lexicographic order, to sort when it encounters letters/characters, so: $a < A < b < B \dots < z < Z$. If you're not sure if a number or a symbol is smaller than another one, you can try it out on the 'Console' in R for example type `'1' < 'a'`. Is this true? Or false? TRUE! How about `'&' > '2'`? FALSE! Is `' ' < '1'` (space less than one)? TRUE!

arrange on several columns

When we include several arguments, it first sorts the first one, then the second one *within* the first order, then the third one *within* the second one, and so on...

```
members %>%  
  arrange(desc(year), citizenship, hired, peak_name)
```

```
## # A tibble: 76,519 x 21
```

##	expedition_id	member_id	peak_id	peak_name	year	season	sex	age
##	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>
##	1	AMAD19107	AMAD1910~	AMAD	Ama Dabl~	2019	Spring M	55
##	2	EVER19116	EVER1911~	EVER	Everest	2019	Spring M	53
##	3	EVER19117	EVER1911~	EVER	Everest	2019	Spring M	38
##	4	EVER19117	EVER1911~	EVER	Everest	2019	Spring M	40
##	5	EVER19148	EVER1914~	EVER	Everest	2019	Spring M	51
##	6	KANG19102	KANG1910~	KANG	Kangchen~	2019	Spring M	43
##	7	EVER19183	EVER1918~	EVER	Everest	2019	Spring M	48
##	8	AMAD19106	AMAD1910~	AMAD	Ama Dabl~	2019	Spring F	33

arrange on transformations

First, let's only take those that got injured, and then see who got furthest away with respect to their injury. Is there anything weird going on? Is my code correct? Is the data correct?

```
members %>%  
  filter(injured) %>%  
  arrange(highpoint_metres - injury_height_metres)
```

```
## # A tibble: 1,713 x 21
```

##		expedition_id	member_id	peak_id	peak_name	year	season	sex	age
##		<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>
##	1	CHOY02327	CHOY0232~	CHOY	Cho Oyu	2002	Autumn	M	29
##	2	MAKA15109	MAKA1510~	MAKA	Makalu	2015	Spring	M	50
##	3	MAKA15109	MAKA1510~	MAKA	Makalu	2015	Spring	M	46
##	4	CHOY03301	CHOY0330~	CHOY	Cho Oyu	2003	Autumn	M	53
##	5	HIML13301	HIML1330~	HIML	Himlung ~	2013	Autumn	M	44
##	6	KANG00106	KANG0010~	KANG	Kangchen~	2000	Spring	M	35

Exercises for arrange:

Exercises for arrange:

- ▶ From those that died, sort by those who got the highest before dying w.r.t. where they died.

Exercises for arrange:

- ▶ From those that died, sort by those who got the highest before dying w.r.t. where they died.
- ▶ Sort by year and then season. Does this make sense? If it doesn't, how could we fix it?

Exercises for arrange:

- ▶ From those that died, sort by those who got the highest before dying w.r.t. where they died.
- ▶ Sort by year and then season. Does this make sense? If it doesn't, how could we fix it?
- ▶ If you arrange by died, can you tell how arrange interprets the logicals?

select

select

This verbs selects the columns that we want to keep. Sometimes, we only need a couple of variables, and it's good to get rid of the rest:

```
members %>%  
  select(year,sex,citizenship)
```

```
## # A tibble: 76,519 x 3  
##   year sex  citizenship  
##   <dbl> <chr> <chr>  
## 1  1978 M      France  
## 2  1978 M      France  
## 3  1978 M      France  
## 4  1978 M      France  
## 5  1978 M      France  
## 6  1978 M      France  
## 7  1978 M      France  
## 8  1978 M      France
```

select ranges

Sometimes we want to select all columns between two other columns, so we can use the colon (:, not the organ) to do this:

select ranges

Sometimes we want to select all columns between two other columns, so we can use the colon (:, not the organ) to do this:

```
members %>%  
  select(age:success)
```

```
## # A tibble: 76,519 x 6
```

```
##      age citizenship expedition_role hired highpoint_metres success  
##    <dbl> <chr>         <chr>         <lgl>         <dbl> <lgl>  
##  1     40 France      Leader      FALSE             NA FALSE  
##  2     41 France    Deputy Leader FALSE          6000 FALSE  
##  3     27 France      Climber     FALSE             NA FALSE  
##  4     40 France    Exp Doctor  FALSE          6000 FALSE  
##  5     34 France      Climber     FALSE             NA FALSE  
##  6     25 France      Climber     FALSE          6000 FALSE  
##  7     41 France      Climber     FALSE          6000 FALSE  
##  8     29 France      Climber     FALSE          6000 FALSE
```

Eliminating with select

Other times we want to get rid of specific variables. For this, we can use the - (minus) symbol.

```
members %>%  
  select(-expedition_id,-member_id,-peak_id)
```

```
## # A tibble: 76,519 x 18
```

```
##   peak_name  year season sex    age citizenship expedition_role hired  
##   <chr>      <dbl> <chr> <chr> <dbl> <chr>          <chr>          <lgl>  
## 1 Ama Dabl~ 1978 Autumn M     40 France      Leader      FALSE  
## 2 Ama Dabl~ 1978 Autumn M     41 France    Deputy Leader FALSE  
## 3 Ama Dabl~ 1978 Autumn M     27 France      Climber      FALSE  
## 4 Ama Dabl~ 1978 Autumn M     40 France    Exp Doctor  FALSE  
## 5 Ama Dabl~ 1978 Autumn M     34 France      Climber      FALSE  
## 6 Ama Dabl~ 1978 Autumn M     25 France      Climber      FALSE  
## 7 Ama Dabl~ 1978 Autumn M     41 France      Climber      FALSE  
## 8 Ama Dabl~ 1978 Autumn M     29 France      Climber      FALSE
```


More of select

We can also use the number of the column to indicate which columns to select, and combine it with the names.

```
members %>%  
  select(4,6:10,highpoint_metres)
```

```
## # A tibble: 76,519 x 7
```

```
##   peak_name    season sex    age citizenship expedition_role highpoint_m
```

```
##   <chr>        <chr> <chr> <dbl> <chr>          <chr>
```

```
## 1 Ama Dablam Autumn M      40 France      Leader
```

```
## 2 Ama Dablam Autumn M      41 France  Deputy Leader
```

```
## 3 Ama Dablam Autumn M      27 France    Climber
```

```
## 4 Ama Dablam Autumn M      40 France  Exp Doctor
```

```
## 5 Ama Dablam Autumn M      34 France    Climber
```

```
## 6 Ama Dablam Autumn M      25 France    Climber
```

```
## 7 Ama Dablam Autumn M      41 France    Climber
```

```
## 8 Ama Dablam Autumn M      29 France    Climber
```

More select functions:

There's a family of functions designed to work with select, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'

More select functions:

There's a family of functions designed to work with select, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'

More select functions:

There's a family of functions designed to work with select, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'
- ▶ `contains('else')` that looks for all the columns that contain 'else' in their name

More select functions:

There's a family of functions designed to work with select, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'
- ▶ `contains('else')` that looks for all the columns that contain 'else' in their name
- ▶ `matches(...)` that looks for all the columns that match with a regular expression (see `?regex`) that we indicate.

More select functions:

There's a family of functions designed to work with select, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'
- ▶ `contains('else')` that looks for all the columns that contain 'else' in their name
- ▶ `matches(...)` that looks for all the columns that match with a regular expression (see `?regex`) that we indicate.
- ▶ `num_range('x', 1:15)` that looks for all columns that are like `x1, x2, ..., x15`

More select functions:

There's a family of functions designed to work with select, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'
- ▶ `contains('else')` that looks for all the columns that contain 'else' in their name
- ▶ `matches(...)` that looks for all the columns that match with a regular expression (see `?regex`) that we indicate.
- ▶ `num_range('x', 1:15)` that looks for all columns that are like `x1, x2, ..., x15`

More select functions:

There's a family of functions designed to work with `select`, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'
- ▶ `contains('else')` that looks for all the columns that contain 'else' in their name
- ▶ `matches(...)` that looks for all the columns that match with a regular expression (see `?regex`) that we indicate.
- ▶ `num_range('x', 1:15)` that looks for all columns that are like `x1`, `x2`, ..., `x15`

Select all the columns that end with 'ed', and the column `solo`. Is there something that stands out?

More select functions:

There's a family of functions designed to work with `select`, so we can work more easily. Among them we have:

- ▶ `starts_with('some')` that looks for all columns that begin with 'some'
- ▶ `ends_with('thing')` that looks for all the columns that start with 'thing'
- ▶ `contains('else')` that looks for all the columns that contain 'else' in their name
- ▶ `matches(...)` that looks for all the columns that match with a regular expression (see `?regex`) that we indicate.
- ▶ `num_range('x', 1:15)` that looks for all columns that are like `x1`, `x2`, ..., `x15`

Select all the columns that end with 'ed', and the column `solo`. Is there something that stands out? Define a dataframe named `members_n` that only has members that died, and no columns end with 'ed' or 'id'.

Renaming with select

We can use select to rename the columns we are choosing:

```
members %>%  
  select(gender = sex, used_oxygen = oxygen_used, role = expedition_role)
```

```
## # A tibble: 76,519 x 3  
##   gender used_oxygen role  
##   <chr>   <lgl>      <chr>  
## 1 M      FALSE      Leader  
## 2 M      FALSE      Deputy Leader  
## 3 M      FALSE      Climber  
## 4 M      FALSE      Exp Doctor  
## 5 M      FALSE      Climber  
## 6 M      FALSE      Climber  
## 7 M      FALSE      Climber  
## 8 M      FALSE      Climber  
## 9 M      FALSE      Climber
```

rename

What did you notice from the previous slide?

rename

What did you notice from the previous slide? We lost all of the columns we didn't mentioned. We can modify this by including everything() next to the last column. Or we can use the function rename.

```
members %>%  
  rename(gender = sex, used_oxygen = oxygen_used, role = expedition_role)
```

```
## # A tibble: 76,519 x 21
```

```
##   expedition_id member_id peak_id peak_name  year season gender  age  
##   <chr>          <chr>    <chr>    <chr>    <dbl> <chr>  <chr>  <dbl>  
## 1 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      40  
## 2 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      41  
## 3 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      27  
## 4 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      40  
## 5 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      34  
## 6 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      25  
## 7 AMAD78301     AMAD7830~ AMAD     Ama Dabl~ 1978 Autumn M      41
```

mutate

mutate



Figure 4: Another type of mutate

mutate

Let's concatenate the role and the citizenship, using the function `paste`, which binds together two strings:

mutate

Let's concatenate the role and the citizenship, using the function `paste`, which binds together two strings:

```
members_n %>%  
  mutate(Role_citizenship = paste(expedition_role,citizenship))
```

```
## # A tibble: 1,106 x 15
```

```
##   peak_name  year season sex    age citizenship expedition_role  
##   <chr>      <dbl> <chr> <chr> <dbl> <chr>          <chr>  
## 1 Ama Dabl~ 1979 Autumn M     23 New Zealand Climber  
## 2 Ama Dabl~ 1983 Autumn M     31 Switzerland Leader  
## 3 Ama Dabl~ 1983 Autumn F     28 Switzerland Climber  
## 4 Ama Dabl~ 1985 Spring M     32 Japan          Climber  
## 5 Ama Dabl~ 1988 Spring M     33 Canada          Climber  
## 6 Ama Dabl~ 1992 Spring M     36 Spain          Leader  
## 7 Annapurn~ 1970 Spring M     32 UK             Climber  
## 8 Annapurn~ 1973 Spring M     37 Japan          Climber
```


mutate

We can also do numeric operations, for example getting differences explicitly:

```
members_n %>%  
  mutate(Difference_mts_died = death_height_metres - highpoint_metres)
```

```
## # A tibble: 1,106 x 15
```

##	peak_name	year	season	sex	age	citizenship	expedition_role
##	<chr>	<dbl>	<chr>	<chr>	<dbl>	<chr>	<chr>
##	1 Ama Dabl~	1979	Autumn	M	23	New Zealand	Climber
##	2 Ama Dabl~	1983	Autumn	M	31	Switzerland	Leader
##	3 Ama Dabl~	1983	Autumn	F	28	Switzerland	Climber
##	4 Ama Dabl~	1985	Spring	M	32	Japan	Climber
##	5 Ama Dabl~	1988	Spring	M	33	Canada	Climber
##	6 Ama Dabl~	1992	Spring	M	36	Spain	Leader
##	7 Annapurn~	1970	Spring	M	32	UK	Climber
##	8 Annapurn~	1973	Spring	M	37	Japan	Climber
##	9 Annapurn~	1973	Spring	M	36	Japan	Climber

mutate to create brand new columns

We can also add and create our own columns, using our own values or from other places. But we have to be very careful they are in the appropriate order, otherwise we risk making a very dangerous mistake. How can it be dangerous?

```
members1905 %>% # this has 9 rows!  
  mutate(my_row = (1:9)^2 + log(15)*(9-row_number()))
```

```
## # A tibble: 9 x 22
```

```
##   expedition_id member_id peak_id peak_name   year season sex      age ci  
##   <chr>          <chr>    <chr>   <chr>    <dbl> <chr> <chr> <dbl> <cl  
## 1 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      29 UK  
## 2 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      NA Sw  
## 3 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      NA Nep  
## 4 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      NA Nep  
## 5 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      NA Nep  
## 6 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      NA Nep  
## 7 KANG05201     KANG0520~ KANG    Kangchen~ 1905 Summer M      36 Sw
```

mutate + if_else

R has a very neat function called `if_else` that is just like an `if`, and checks whether a condition is true, then do something, if it's not, then do something else:

```
members1905 %>% ## notice the difference between: ' and "  
  mutate(Cheating = if_else(expedition_role == 'Leader',  
                             'Cheated',  
                             "Didn't cheat"))
```

```
## # A tibble: 9 x 22
```

```
##   expedition_id member_id peak_id peak_name   year season sex    age ci  
##   <chr>          <chr>    <chr>  <chr>    <dbl> <chr> <chr> <dbl> <chr>  
## 1 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      29 UK  
## 2 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Sw  
## 3 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Ne  
## 4 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Ne  
## 5 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Ne  
## 6 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Ne  
## 7 KANG05201     KANG0520~ KANG   Kangchen~ 1905 Summer M      NA Ne
```

mutate + other functions

We can also combine mutate with other functions to obtain new columns that depend on all the values from specific columns of the dataframe:

```
members1905 %>%  
  mutate(anyone_died = any(died),  
         max_height = max(highpoint_metres, na.rm = TRUE),  
         last_citizenship = last(citizenship),  
         youngest = min(age),  
         percent_dead = mean(died),  
         diff_from_average_height_death = death_height_metres - mean(death_h  
  select(-(1:21)) # to see the new variables
```

```
## # A tibble: 9 x 6
```

```
##   anyone_died max_height last_citizenship youngest percent_dead diff_fr
```

```
##   <lgl>          <dbl> <chr>                <dbl>          <dbl>
```

```
## 1 TRUE          6300 Switzerland             NA          0.556
```

```
## 2 TRUE          6300 Switzerland             NA          0.556
```

```
## 3 TRUE          6300 Switzerland             NA          0.556
```

Exercises mutate

Exercises mutate

- ▶ What's the percentage of people that died out of those that got injured?

Exercises mutate

- ▶ What's the percentage of people that died out of those that got injured?
- ▶ Out of those that died, how many got injured before?

Exercises mutate

- ▶ What's the percentage of people that died out of those that got injured?
- ▶ Out of those that died, how many got injured before?
- ▶ Create a new column that is the concatenation of the member id and it's citizenship.

summarise

summarise

It's just like mutate, but the output has to be of only one row, and it eliminates (in the output) everything else that is not mentioned*:

summarise

It's just like mutate, but the output has to be of only one row, and it eliminates (in the output) everything else that is not mentioned*:

```
members1905 %>%  
  summarise(anyone_died = any(died),  
            max_height = max(highpoint_metres, na.rm = TRUE),  
            last_citizenship = last(citizenship),  
            youngest = min(age),  
            percent_dead = mean(died))
```

```
## # A tibble: 1 x 5  
##   anyone_died max_height last_citizenship youngest percent_dead  
##   <lgl>         <dbl> <chr>                <dbl>         <dbl>  
## 1 TRUE         6300 Switzerland          NA           0.556
```

We are only getting one row, and losing everything else. Using, filter, mutate and summarise, indicate how far the average of the highpoint metres for the women from France differs from the average from all the table (use na.rm=TRUE)

group_by

group_by



Figure 5: Another type of group

group_by

```
members %>%  
  group_by(died) %>%  
  summarise(Percentage_injured = mean(injured))
```

```
## 'summarise()' ungrouping output (override with '.groups' argument)
```

```
## # A tibble: 2 x 2  
##   died Percentage_injured  
##   <lgl>                <dbl>  
## 1 FALSE                0.0227  
## 2 TRUE                 0.00271
```

What do you think group_by does?

group_by

On it's own, group_by doesn't do much, it really shines when we combine it with the other 5 verbs that we have studied.

group_by

On it's own, `group_by` doesn't do much, it really shines when we combine it with the other 5 verbs that we have studied.

All it does is group by the variables we tell it to, and the following modifications that happen on the `data.frame` are done on each of the groups we defined, as if each group was a `dataframe`.

group_by + mutate

When we combine it with mutate, we get a new observation, that is only different for each of the grouped variables.

```
members_n %>% select(-(4:14)) %>%  
  group_by(season) %>%  
  mutate(Obs_per_season = n())
```

```
## # A tibble: 1,106 x 4  
## # Groups:   season [4]  
##   peak_name    year season Obs_per_season  
##   <chr>      <dbl> <chr>         <int>  
## 1 Ama Dablam  1979 Autumn         493  
## 2 Ama Dablam  1983 Autumn         493  
## 3 Ama Dablam  1983 Autumn         493  
## 4 Ama Dablam  1985 Spring         555  
## 5 Ama Dablam  1988 Spring         555  
## 6 Ama Dablam  1992 Spring         555
```

`group_by + mutate`

Repeat what we did above, but grouping by year, instead of season.

More than one group

More than one group

If you want to group with more than one variable, you can simply add the column in the argument:

```
members %>%  
  group_by(year,season) %>%  
  summarise(Average_height = mean(hightpoint_metres,  
                                   na.rm = TRUE))
```

```
## 'summarise()' regrouping output by 'year' (override with '.groups' argument)
```

```
## # A tibble: 239 x 3
```

```
## # Groups:   year [92]
```

```
##   year season Average_height
```

```
##   <dbl> <chr>         <dbl>
```

```
## 1  1905 Summer         6300
```

```
## 2  1907 Autumn         7270
```

```
## 3  1909 Autumn         6965
```

```
## 4  1910 Spring         6965
```

group_by exercises

group_by exercises

- ▶ What year had the most dead people? (Hint: arrange)

group_by exercises

- ▶ What year had the most dead people? (Hint: arrange)
- ▶ Using only those with a role of climbers, compute the mean and standard deviation of the yearly number of climbers for each season. (Hint: use season as the first grouping variable)

group_by exercises

- ▶ What year had the most dead people? (Hint: arrange)
- ▶ Using only those with a role of climbers, compute the mean and standard deviation of the yearly number of climbers for each season. (Hint: use season as the first grouping variable)
- ▶

group_by + summarize

Answer to the second question above:

```
members %>%  
  filter(expedition_role == 'Climber') %>%  
  group_by(season, year) %>%  
  summarize(Total = n()) %>%  
  summarise(Mean = mean(Total),  
            Std = sd(Total))
```

```
## 'summarise()' regrouping output by 'season' (override with '.groups' arg
```

```
## 'summarise()' ungrouping output (override with '.groups' argument)
```

```
## # A tibble: 5 x 3
```

```
##   season   Mean   Std
```

```
##   <chr>   <dbl> <dbl>
```

```
## 1 Autumn  321.  270.
```

```
## 2 Spring  256.  240.
```

group_by + filter

If you want to eliminate the groups that don't have enough, or that have too many observations, you can do it by combining filter and group_by directly:

```
(surviving_members <- members %>%  
  group_by(expedition_id) %>%  
  filter(n() > 10))
```

```
## # A tibble: 38,762 x 21
```

```
## # Groups:   expedition_id [2,264]
```

	expedition_id	member_id	peak_id	peak_name	year	season	sex	age
	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>
##	1	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	35
##	2	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	37
##	3	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	23
##	4	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	44
##	5	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	25
##	6	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	28
##	7	AMAD79101	AMAD7910~	AMAD	Ama Dabl~	1979	Spring M	22

rowwise

This function, as its name tells us, is like doing a `group_by`, but operates on each row. This one is particularly useful when you are creating your own functions and they have *weird* interactions with vectors, like using `sum`, `mean`, and such... But we would probably get different results on each row.

Thank you :)

Any questions?

Further references

- ▶ Check the help page and the vignettes of dplyr! (type `?dplyr`, or: `vignette('dplyr')` on the console)
- ▶ R for Data Science, by Hadley Wickham
- ▶ Advanced R, by Hadley Wickham
- ▶ The R Inferno, by Patrick Burns