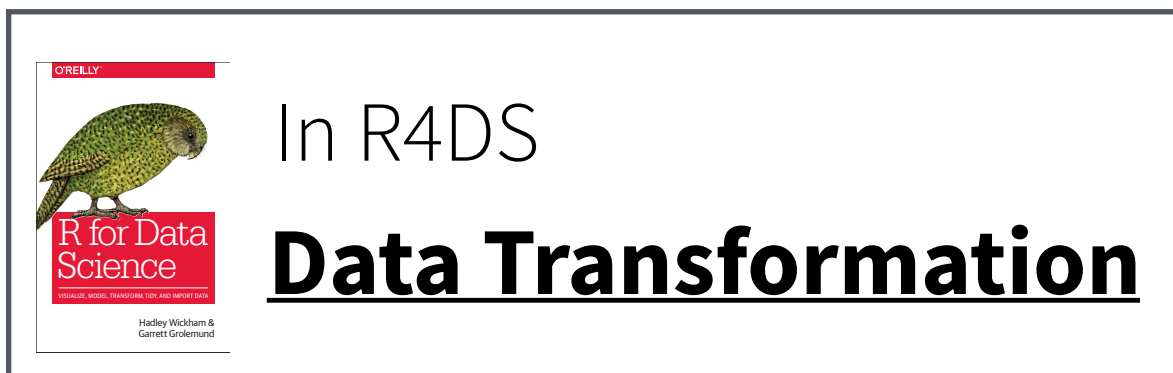


# Transform Data with

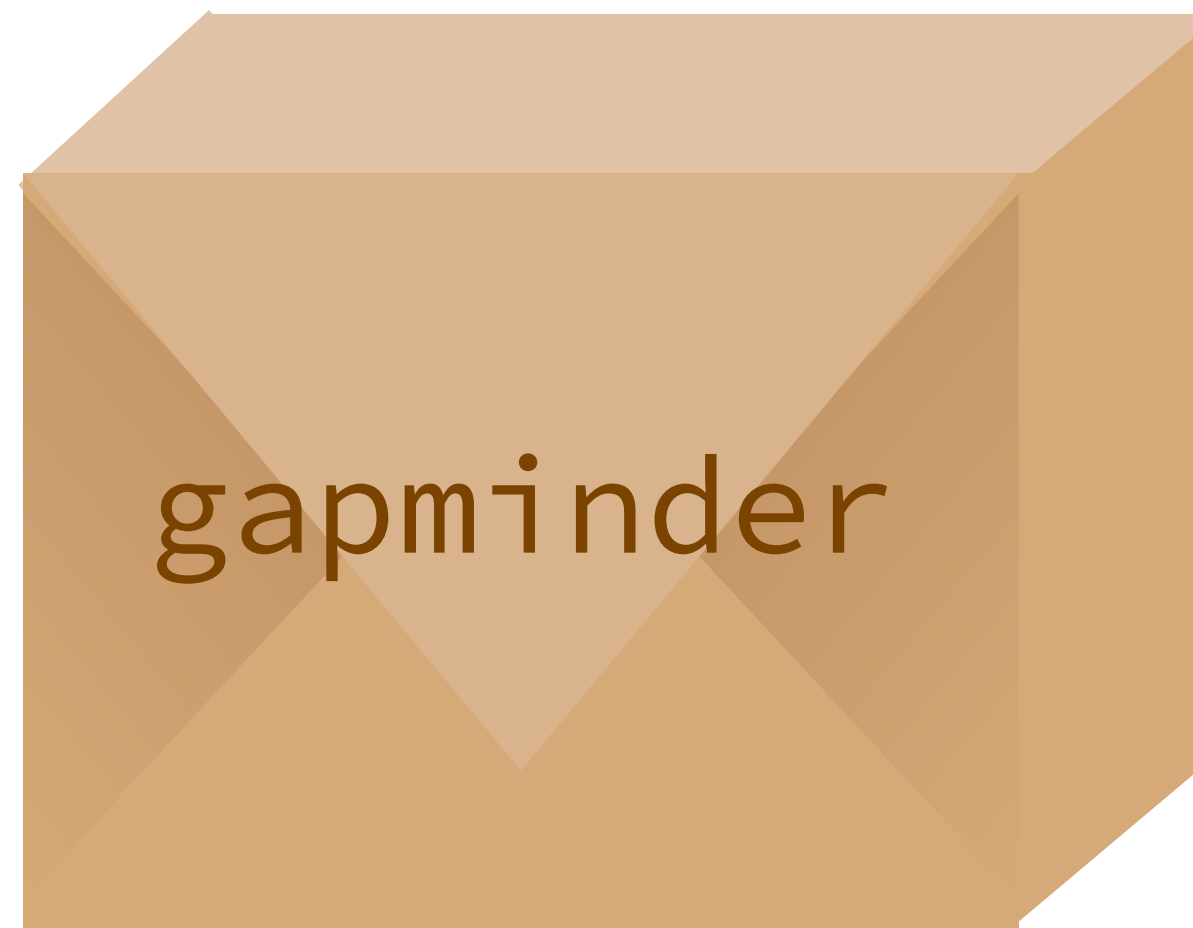


# Your Turn 0

1. Open 02-transform.Rmd
2. Run the setup chunk

02:00

# gapminder



A subset of Gapminder data: population, GDP per capita and life expectancy, for countries over time.

```
# install.packages("gapminder")  
library(gapminder)
```





# gapminder

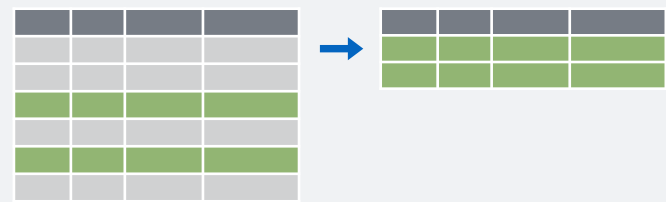
country <fctr>	continent <fctr>	year <int>	lifeExp <dbl>	pop <int>	gdpPercap <dbl>
Afghanistan	Asia	1952	28.80100	8425333	779.4453
Afghanistan	Asia	1957	30.33200	9240934	820.8530
Afghanistan	Asia	1962	31.99700	10267083	853.1007
Afghanistan	Asia	1967	34.02000	11537966	836.1971
Afghanistan	Asia	1972	36.08800	13079460	739.9811
Afghanistan	Asia	1977	38.43800	14880372	786.1134
Afghanistan	Asia	1982	39.85400	12881816	978.0114
Afghanistan	Asia	1987	40.82200	13867957	852.3959
Afghanistan	Asia	1992	41.67400	16317921	649.3414
Afghanistan	Asia	1997	41.76300	22227415	635.3414

1-10 of 1,704 rows

Previous 1 2 3 4 5 6 ... 100 Next



# dplyr: Data manipulation



Extract cases with **filter()**



Extract variables with **select()**



Arrange cases, with **arrange()**.



Make new variables, with **mutate()**.



Make tables of summaries with **summarise()**.

along with **group\_by()**



`filter()`

# 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)



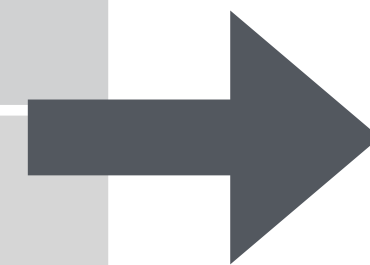
# filter()

Extract rows that meet logical criteria.

```
filter(gapminder, country == "New Zealand")
```

gapminder

country	continent	year	...
Afghanistan	Asia	1952	
Afghanistan	Asia	1957	
...	...	...	
Netherlands	Europe	2007	
New Zealand	Oceania	1952	
New Zealand	Oceania	1957	



country	continent	year	...
New Zealand	Oceania	1952	
New Zealand	Oceania	1957	
New Zealand	Oceania	1962	
New Zealand	Oceania	1967	
...	...	...	...





# filter()

Extract rows that meet logical criteria.

```
filter(gapminder, country == "New Zealand")
```

**= sets**

(returns nothing)

**== tests if equal**

(returns TRUE or FALSE)



# Logical tests

## ?Comparison

<code>x &lt; y</code>	Less than
<code>x &gt; y</code>	Greater than
<code>x == y</code>	Equal to
<code>x &lt;= y</code>	Less than or equal to
<code>x &gt;= 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



# Your Turn 1

See if you can use the logical operators to manipulate our code below to show:

1. The data for United States
2. All data for countries in Oceania
3. Rows where the life expectancy is greater than 82

04:00

```
filter(gapminder, country == "United States")
```

```
filter(gapminder, continent == "Oceania")
```

```
filter(gapminder, lifeExp > 82)
```

# Two common mistakes

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

```
filter(gapminder, continent = "Oceania")  
filter(gapminder, continent == "Oceania")
```

## 2. Forgetting quotes

```
filter(gapminder, continent == Oceania)  
filter(gapminder, continent == "Oceania")
```





# filter()

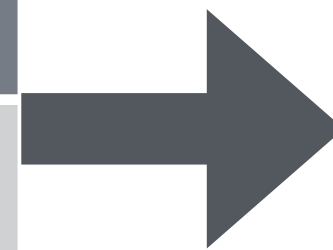
additional  
arguments must  
also be TRUE

Extract rows that meet *every* logical criteria.

```
filter(gapminder, country == "New Zealand", year > 2000)
```

gapminder

country	continent	year	...
...	...	...	
New Zealand	Oceania	1952	
...	...	...	
New Zealand	Oceania	2002	
New Zealand	Oceania	2007	



country	continent	year	...
New Zealand	Oceania	2002	
New Zealand	Oceania	2007	

# Boolean operators

?base::Logic

<code>a &amp; b</code>	and
<code>a   b</code>	or
<code>!a</code>	not

# filter()

Extract rows that meet *every* logical criteria.

```
filter(gapminder, country == "New Zealand" & year > 2000)
```

gapminder

country	continent	year	...
...	...	...	
New Zealand	Oceania	1952	
...	...	...	
New Zealand	Oceania	2002	
New Zealand	Oceania	2007	

country	continent	year	...
New Zealand	Oceania	2002	
New Zealand	Oceania	2007	

# Your Turn 2

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

1. United States before 1970
2. Countries where life expectancy in 2007 is below 50
3. Records for any of "New Zealand", "Canada" or "United States"

04:00

```
filter(gapminder, country == "Canada", year < 1970)
```

```
filter(gapminder, year == 2007, lifeExp < 50)
```

```
filter(gapminder,  
  country %in% c("Canada", "New Zealand", "United States"))
```



# Two more common mistakes

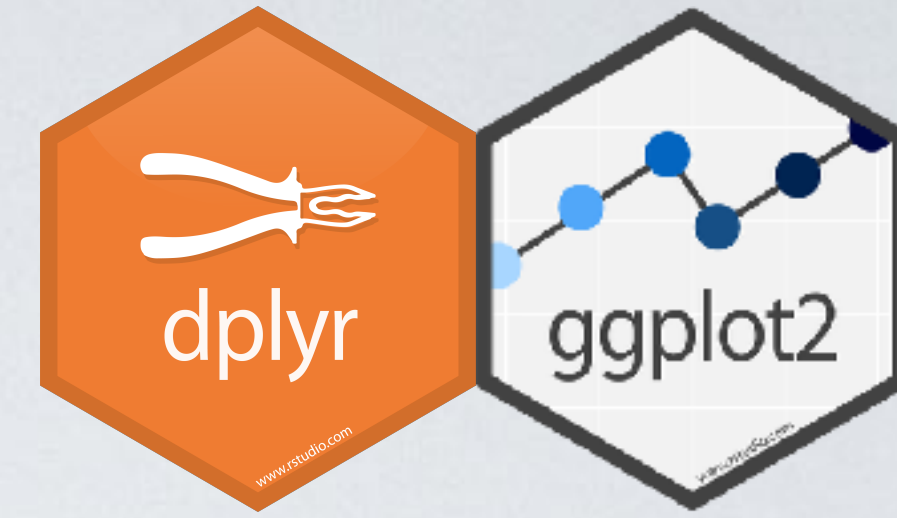
## 3. Collapsing multiple tests into one

```
filter(gapminder, 1960 < year < 1980)  
filter(gapminder, 1960 < year, year < 1980)
```

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

```
filter(gapminder, country == "New Zealand" |  
  country == "Canada" | country == "United States")  
filter(gapminder,  
  country %in% c("New Zealand", "Canada", "United States"))
```

# Your Turn 3



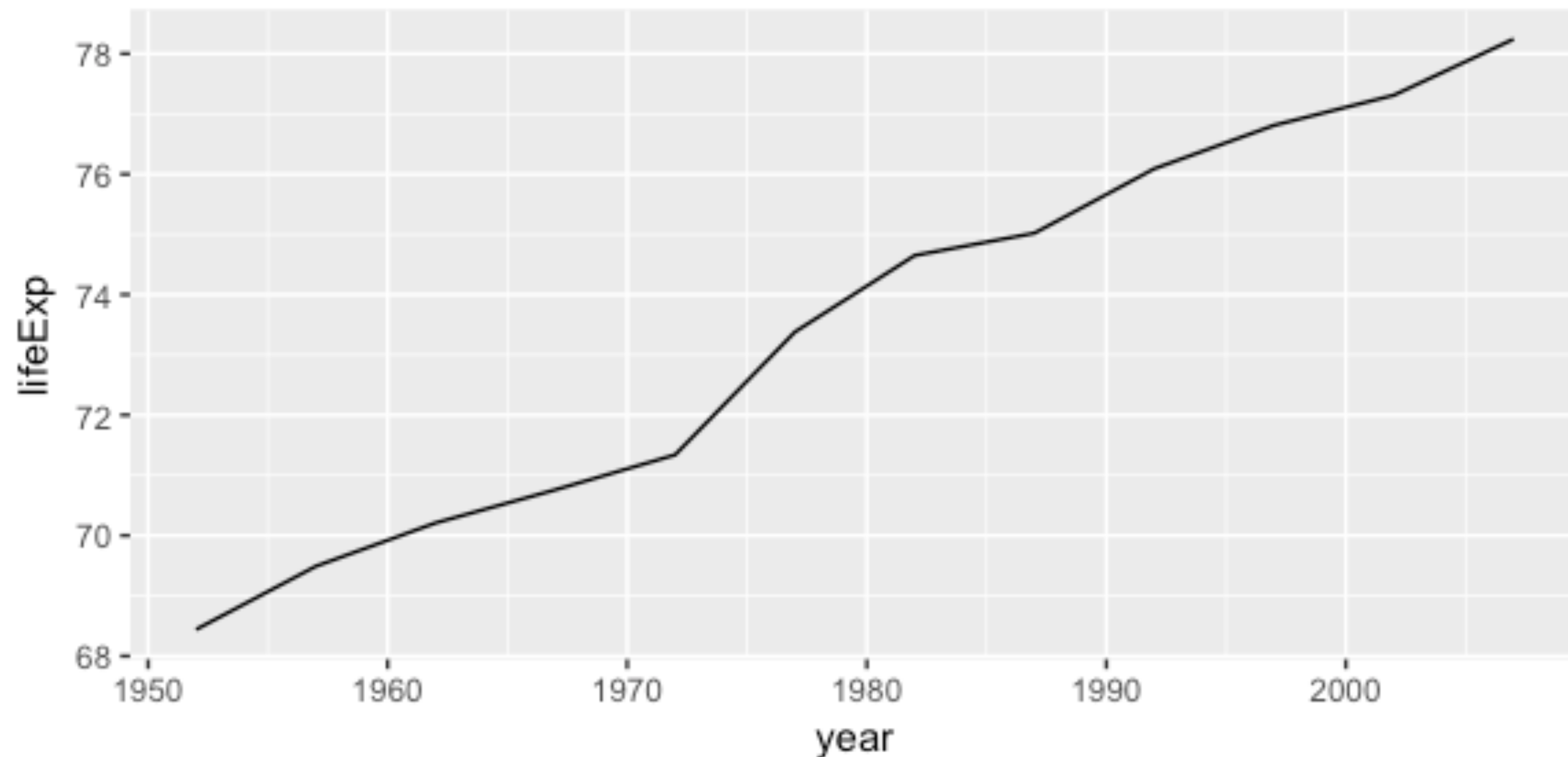
Use `filter()` to get the records for the US, then plot the life expectancy over time.

04:00

```
filter(gapminder, country == "United States")
```

```
us <- filter(gapminder, country == "United States")
```

```
ggplot(us, aes(x = year, y = lifeExp)) +  
  geom_line()
```



# dplyr 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**





arrange()

# arrange()

Order cases from smallest to largest values.

```
arrange(.data, ...)
```

**data frame to  
transform**

**one or more columns to order by**  
(additional columns will be used as  
tie breakers)



# arrange()

Order cases from smallest to largest values.

```
arrange(gapminder, lifeExp)
```

country	continent	year	lifeExp	...	country	continent	year	lifeExp	...
Afghanista	Afghanistan	1952	28.8		Rwanda	Rwanda	1992	23.6	
Afghanista	Afghanistan	1957	30.3		Afghanista	Afghanistan	1952	28.8	
Angola	Angola	1952	30.0		Gambia	Gambia	1952	30.0	
Gambia	Gambia	1952	30.0		Angola	Angola	1952	30.0	
Rwanda	Rwanda	1992	23.6		Sierra	Sierra	1952	30.3	
Sierra	Sierra	1952	30.3		Afghanista	Afghanistan	1957	30.3	

# desc()

Changes order to largest to smallest values.

```
arrange(gapminder, desc(lifeExp))
```

country	continent	year	lifeExp	...
Afghanista	Afghanistan	1952	28.8	
Afghanista	Afghanistan	1957	30.3	
Angola	Angola	1952	30.0	
Gambia	Gambia	1952	30.0	
Rwanda	Rwanda	1992	23.6	
Sierra	Sierra	1952	30.3	

country	continent	year	lifeExp	...
Afghanista	Afghanistan	1957	30.3	
Sierra	Sierra	1952	30.3	
Angola	Angola	1952	30.0	
Gambia	Gambia	1952	30.0	
Afghanista	Afghanistan	1952	28.8	
Rwanda	Rwanda	1992	23.6	

# Your Turn 4

Find the records with the smallest population.

Find the records with the largest GDP per capita.

02:00



```
arrange(gapminder, pop)
```

```
## A tibble: 1,704 x 6
```

#	country	conti...	year	lifeE...	pop	gdpP...
#	<fctr>	<fctr>	<int>	<dbl>	<int>	<dbl>
# 1	Sao Tome and Principe	Africa	1952	46.5	60011	880
# 2	Sao Tome and Principe	Africa	1957	48.9	61325	861
# 3	Djibouti	Africa	1952	34.8	63149	2670
# 4	Sao Tome and Principe	Africa	1962	51.9	65345	1072
# 5	Sao Tome and Principe	Africa	1967	54.4	70787	1385

```
arrange(gapminder, desc(gdpPercap))
```

```
## A tibble: 1,704 x 6
```

```
#   country    continent  year lifeExp      pop gdpPercap
#   <fctr>      <fctr>    <int>  <dbl>    <int>    <dbl>
# 1 Kuwait      Asia      1957   58.0  212846  113523
# 2 Kuwait      Asia      1972   67.7  841934  109348
# 3 Kuwait      Asia      1952   55.6  160000  108382
# 4 Kuwait      Asia      1962   60.5  358266   95458
# 5 Kuwait      Asia      1967   64.6  575003   80895
```

Pipe %>%

# Multistep Operations

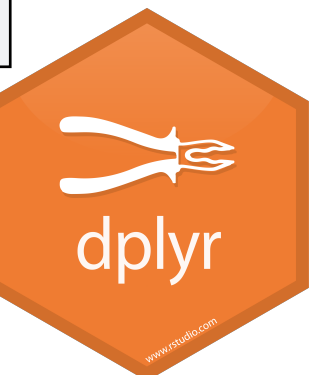
Consider the following:

**Extract rows** with year equal to 2007, then

**Arrange** by decreasing life expectancy

**Option 1:** Use intermediate variables

```
gapminder_2007 <- filter(gapminder, year == 2007)
arrange(gapminder_2007, desc(lifeExp))
```



# Multistep Operations

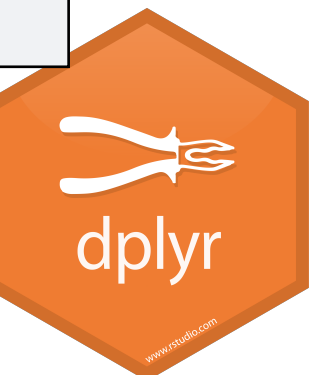
Consider the following:

**Extract rows** with year equal to 2007, then

**Arrange** by decreasing life expectancy

**Option 2:** Do it all in one line

```
arrange(filter(gapminder, year == 2007), desc(lifeExp))
```





# Multistep Operations

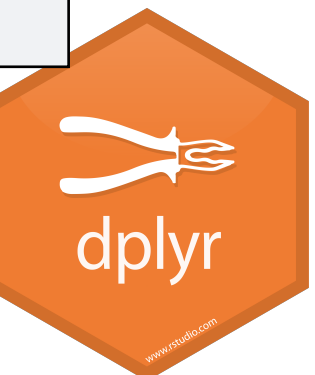
Consider the following:

**Extract rows** with year equal to 2007, then

**Arrange** by decreasing life expectancy

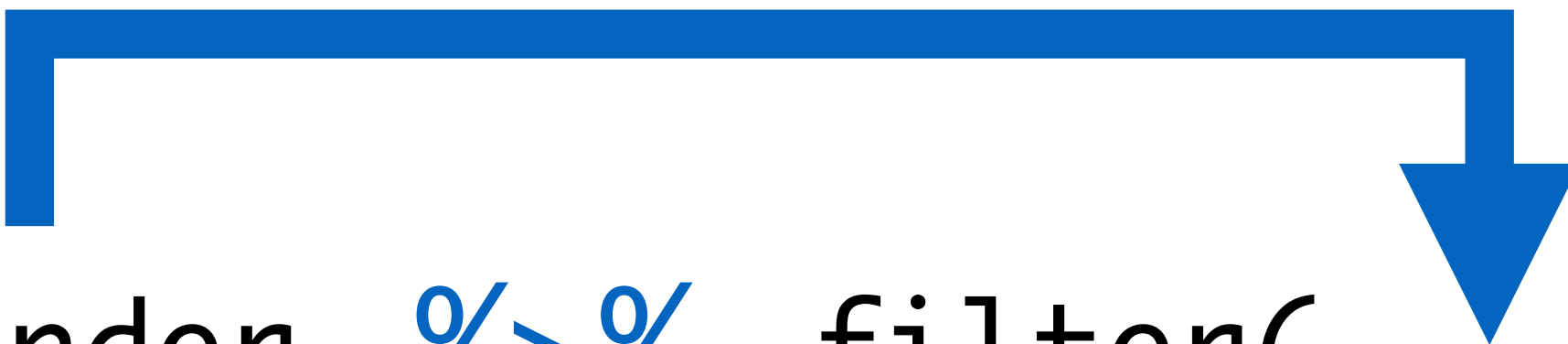
**Option 2:** Do it all in one line

```
arrange(filter(gapminder, year == 2007), desc(lifeExp))
```



# The pipe operator %>%

Passes result on left into first argument of function on right.



```
gapminder %>% filter(____, country == "Canada")
```

These do the same thing. Try it.

```
filter(gapminder, country == "Canada")  
gapminder %>% filter(country == "Canada")
```



# Multistep Operations

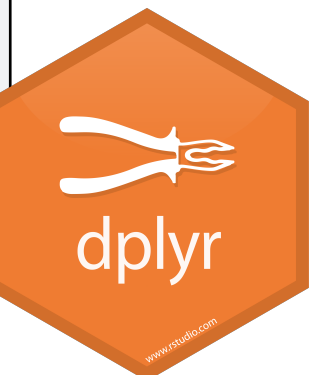
Consider the following:

**Extract rows** with year equal to 2007, then

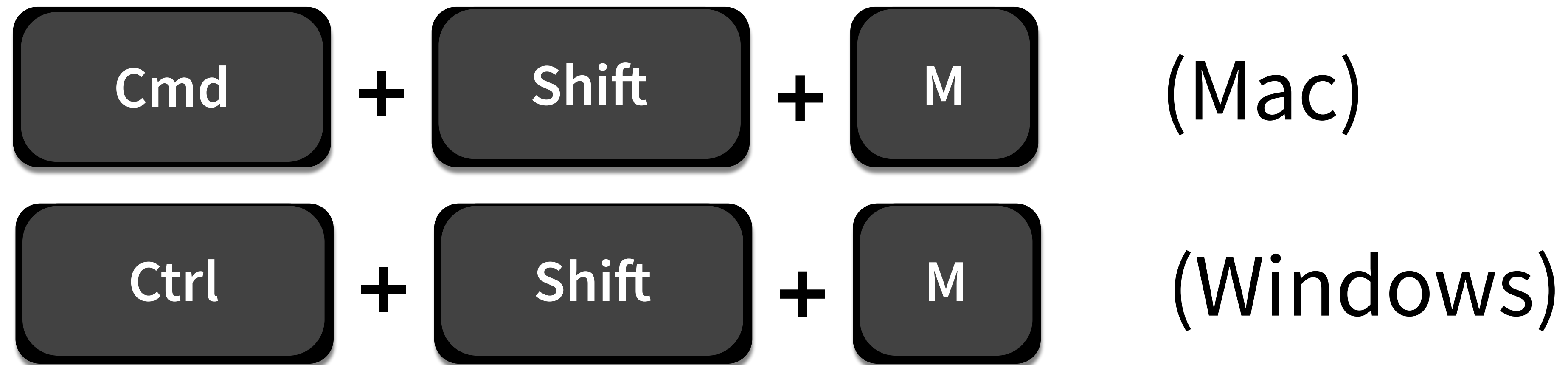
**Arrange** by decreasing life expectancy

**Option 3:** Use the pipe

```
gapminder %>%  
  filter(year == 2007) %>%  
  arrange(desc(lifeExp))
```



# Shortcut to type %>%



**mutate()**



# mutate()

Create new columns.

```
gapminder %>% mutate(gpd = gdpPercap * pop)
```



# mutate()

Create new columns.

```
gapminder %>% mutate(gpd = gdpPercap * pop)
```

dplyr function

data frame to  
transform

function specific  
arguments



# mutate()

Create new columns.

```
gapminder %>% mutate(gpd = gdpPercap * pop)
```

argument name is  
new column name

argument value is how  
to calculate, based on  
existing columns



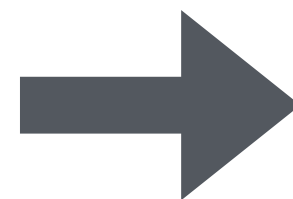
# mutate()

Create new columns.

```
gapminder %>% mutate(gpd = gdpPercap * pop)
```

gapminder

country	continent	year	...
Afghanistan	Asia	1952	
Afghanistan	Asia	1957	
Afghanistan	Asia	1962	
Afghanistan	Asia	1967	
Afghanistan	Asia	1972	
Afghanistan	Asia	1977	



country	continent	year	...	gdp
Afghanistan	Asia	1952		6567086330
Afghanistan	Asia	1957		7585448670
Afghanistan	Asia	1962		8758855797
Afghanistan	Asia	1967		9648014150
Afghanistan	Asia	1972		9678553274
Afghanistan	Asia	1977		11697659231



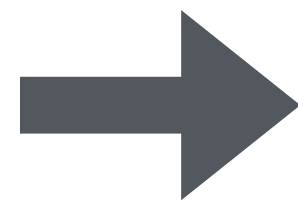
# mutate()

Create new columns.

```
gapminder %>% mutate(gdp = gdpPercap * pop,  
                     pop_mill = round(pop/1000000))
```

gapminder

country	continent	year	...
Afghanistan	Asia	1952	
Afghanistan	Asia	1957	
Afghanistan	Asia	1962	
Afghanistan	Asia	1967	
Afghanistan	Asia	1972	
Afghanistan	Asia	1977	



country	continent	year	...	gdp	pop_mill
Afghanistan	Asia	1952		6567086330	8
Afghanistan	Asia	1957		7585448670	9
Afghanistan	Asia	1962		8758855797	10
Afghanistan	Asia	1967		9648014150	12
Afghanistan	Asia	1972		9678553274	13
Afghanistan	Asia	1977		11697659231	15



# Quiz

A function that returns a vector the same length as the input is called **vectorized**.

Which of the following functions are vectorized?

- `ifelse()`
- `diff()`
- `sum()`

02:00

```
gapminder %>%  
  mutate(size = ifelse(pop < 10e06, "small", "large"))
```

OK, **ifelse()** is  
vectorized

```
gapminder %>%  
  mutate(diff_pop = diff(pop))
```

Error in mutate\_impl(.data, dots) :  
Column `diff\_pop` must be length 1704  
(the number of rows) or one, not 1703

Not OK, **diff()** is **not**  
vectorized



```
gapminder %>%
```

```
  mutate(total_pop = sum(as.numeric(pop)))
```

```
# A tibble: 1,704 x 7
```

	country	continent	year	lifeExp	pop	gdpPercap	total_pop
	<fctr>	<fctr>	<int>	<dbl>	<int>	<dbl>	<dbl>
1	Afghanistan	Asia	1952	28.8	8425333	779	50440465801
2	Afghanistan	Asia	1957	30.3	9240934	821	50440465801
3	Afghanistan	Asia	1962	32.0	10267083	853	50440465801

OK, **sum()** is **not** vectorized,

but **mutate()** just repeats the single  
returned values to fill the rows.





## Vectorized Functions

### TO USE WITH MUTATE ()

**mutate()** and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

#### vectorized function

### OFFSETS

**dplyr::lag()** - Offset elements by 1  
**dplyr::lead()** - Offset elements by -1

### CUMULATIVE AGGREGATES

**dplyr::cumall()** - Cumulative all()  
**dplyr::cumany()** - Cumulative any()  
**cummax()** - Cumulative max()  
**dplyr::cummean()** - Cumulative mean()  
**cummin()** - Cumulative min()  
**cumprod()** - Cumulative prod()  
**cumsum()** - Cumulative sum()

### RANKINGS

**dplyr::cume\_dist()** - Proportion of all values <=  
**dplyr::dense\_rank()** - rank with ties = min, no gaps  
**dplyr::min\_rank()** - rank with ties = min  
**dplyr::ntile()** - bins into n bins  
**dplyr::percent\_rank()** - min\_rank scaled to [0,1]  
**dplyr::row\_number()** - rank with ties = "first"

### MATH

**+**, **-**, **\***, **/**, **^**, **%/%**, **%%** - arithmetic ops  
**log()**, **log2()**, **log10()** - logs  
**<**, **<=**, **>**, **>=**, **!=**, **==** - logical comparisons

### MISC

**dplyr::between()** -  $x \geq \text{left} \ \& \ x \leq \text{right}$   
**dplyr::case\_when()** - multi-case if\_else()  
**dplyr::coalesce()** - first non-NA values by element across a set of vectors  
**dplyr::if\_else()** - element-wise if() + else()  
**dplyr::na\_if()** - replace specific values with NA  
**pmax()** - element-wise max()  
**pmin()** - element-wise min()  
**dplyr::recode()** - Vectorized switch()  
**dplyr::recode\_factor()** - Vectorized switch() for factors

# Vectorized functions

Take a vector as input.

Return a vector of the same length as output.

## Vectorized Functions

### TO USE WITH MUTATE ()

**mutate()** and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

#### vectorized function

### OFFSETS

**dplyr::lag()** - Offset elements by 1  
**dplyr::lead()** - Offset elements by -1

### CUMULATIVE AGGREGATES

**dplyr::cumall()** - Cumulative all()  
**dplyr::cumany()** - Cumulative any()  
**cummax()** - Cumulative max()  
**dplyr::cummean()** - Cumulative mean()  
**cummin()** - Cumulative min()  
**cumprod()** - Cumulative prod()  
**cumsum()** - Cumulative sum()

### RANKINGS

**dplyr::cume\_dist()** - Proportion of all values <=  
**dplyr::dense\_rank()** - rank with ties = min, no gaps  
**dplyr::min\_rank()** - rank with ties = min  
**dplyr::ntile()** - bins into n bins  
**dplyr::percent\_rank()** - min\_rank scaled to [0,1]  
**dplyr::row\_number()** - rank with ties = "first"

### MATH

**+**, **-**, **\***, **/**, **^**, **%/%**, **%%** - arithmetic ops  
**log()**, **log2()**, **log10()** - logs  
**<**, **<=**, **>**, **>=**, **!=**, **==** - logical comparisons

### MISC

**dplyr::between()** -  $x \geq \text{left} \ \& \ x \leq \text{right}$   
**dplyr::case\_when()** - multi-case if\_else()  
**dplyr::coalesce()** - first non-NA values by element across a set of vectors  
**dplyr::if\_else()** - element-wise if() + else()  
**dplyr::na\_if()** - replace specific values with NA  
**pmax()** - element-wise max()  
**pmin()** - element-wise min()  
**dplyr::recode()** - Vectorized switch()  
**dplyr::recode\_factor()** - Vectorized switch() for factors



## Summary Functions

### TO USE WITH SUMMARISE ()

**summarise()** applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

#### summary function

### COUNTS

**dplyr::n()** - number of values/rows  
**dplyr::n\_distinct()** - # of unique  
**sum(!is.na())** - # of non-NA's

### LOCATION

**mean()** - mean, also **mean(!is.na())**  
**median()** - median

### LOGICALS

**mean()** - Proportion of TRUE's  
**sum()** - # of TRUE's

### POSITION/ORDER

**dplyr::first()** - first value  
**dplyr::last()** - last value  
**dplyr::nth()** - value in nth location of vector

### RANK

**quantile()** - nth quantile  
**min()** - minimum value  
**max()** - maximum value

### SPREAD

**IQR()** - Inter-Quartile Range  
**mad()** - mean absolute deviation  
**sd()** - standard deviation  
**var()** - variance

## Row Names

Many data does not use rownames, which stores a variable outside of the columns. To work with the rownames, first move them into a column.

**rownames\_to\_column()**  
Move row names into col.  
**a <- rownames\_to\_column(fis, var = "r")**

**column\_to\_rownames()**  
Move col in row names.  
**column\_to\_rownames(fu, var = "C")**

Also has **rownames()**, **remove\_rownames()**

## Combine Tables

### COMBINE VARIABLES

Use **bind\_cols()** to paste tables beside each other as they are.

**bind\_cols(...)** Returns tables placed side by side as a single table.  
BE SURE THAT ROWS ALIGN.

Use a "Mutating Join" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

**left\_join(x, y, by = NULL, copy = FALSE, suffix = c("x", "y"), ...)**  
Join matching values from y to x.

**right\_join(x, y, by = NULL, copy = FALSE, suffix = c("x", "y"), ...)**  
Join matching values from x to y.

**inner\_join(x, y, by = NULL, copy = FALSE, suffix = c("x", "y"), ...)**  
Join data. Retain only rows with matches.

**full\_join(x, y, by = NULL, copy = FALSE, suffix = c("x", "y"), ...)**  
Join data. Retain all values, all rows.

Use **by = c("col1", "col2")** to specify the column(s) to match on.  
**left\_join(x, y, by = "A")**

Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.  
**left\_join(x, y, by = c("C" = "D"))**

Use **suffix** to specify suffix to give to duplicate column names.  
**left\_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))**

### COMBINE CASES

Use **bind\_rows()** to paste tables below each other as they are.

**bind\_rows(..., id = NULL)**  
Returns tables one on top of the other as a single table. Set **id** to a column name to add a column of the original table names (as pictured).

**intersect(x, y, ...)**  
Rows that appear in both x and y.

**setdiff(x, y, ...)**  
Rows that appear in x but not y.

**union(x, y, ...)**  
Rows that appear in x or y.  
(Duplicates removed). **union\_all()** retains duplicates.

Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

Use a "Filtering Join" to filter one table against the rows of another.

**semi\_join(x, y, by = NULL, ...)**  
Return rows of x that have a match in y. USEFUL TO SEE WHAT WILL BE JOINED.

**anti\_join(x, y, by = NULL, ...)**  
Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

Most useful:

- Math
- Misc





# Your Turn 5

Alter the code to add a `prev_lifeExp` column that contains the life expectancy from the previous record. (**Hint:** use the cheatsheet, you want to offset elements by one)

**Extra challenge:** Why isn't this quite '*life expectancy five years ago*'?

03:00



```
gapminder %>%
```

```
  mutate(prev_lifeExp = lag(lifeExp))
```

country <fctr>	continent <fctr>	year <int>	lifeExp <dbl>	pop <int>	gdpPercap <dbl>	prev_lifeExp <dbl>
Afghanistan	Asia	1952	28.80100	8425333	779.4453	NA
Afghanistan	Asia	1957	30.33200	9240934	820.8530	28.80100
Afghanistan	Asia	1962	31.99700	10267083	853.1007	30.33200
Afghanistan	Asia	1967	34.02000	11537966	836.1971	31.99700
Afghanistan	Asia	1972	36.08800	13079460	739.9811	34.02000
Afghanistan	Asia	1977	38.43800	14880372	786.1134	36.08800
Afghanistan	Asia	1982	39.85400	12881816	978.0114	38.43800
Afghanistan	Asia	1987	40.82200	13867957	852.3959	39.85400
Afghanistan	Asia	1992	41.67400	16317921	649.3414	40.82200
Afghanistan	Asia	1997	41.76300	22227415	635.3414	41.67400

1-10 of 1,704 rows

Previous 1 2 3 4 5 6 ... 100 Next



This is the life expectancy for Afghanistan 55 years in the future

country <fctr>	continent <fctr>	year <int>	lifeExp <dbl>	pop <int>	gdpPercap <dbl>	
Afghanistan	Asia	2002	42.12900	25268405	726.7341	
Afghanistan	Asia	2007	43.82800	31889923	974.5803	42.12900
Albania	Europe	1952	55.23000	1282697	1601.0561	43.82800
Albania	Europe	1957	59.28000	1476505	1942.2842	55.23000
Albania	Europe	1962	64.82000	1728137	2312.8890	59.28000
Albania	Europe	1967	66.22000	1984060	2760.1969	64.82000
Albania	Europe	1972	67.69000	2263554	3313.4222	66.22000
Albania	Europe	1977	68.93000	2509048	3533.0039	67.69000
Albania	Europe	1982	70.42000	2780097	3630.8807	68.93000
Albania	Europe	1987	72.00000	3075321	3738.9327	70.42000

11-20 of 1,704 rows

Previous 1 2 3 4 5 6 ... 100 Next

**summarise()**

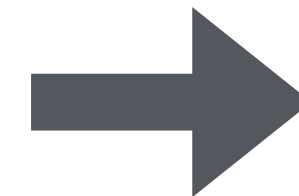
# summarise()

Compute table of summaries.

```
gapminder %>% summarise(mean_life = mean(lifeExp))
```

gapminder

country	continent	year	lifeExp	...
Afghanistan	Asia	1952	28.801	
Afghanistan	Asia	1957	30.332	
Afghanistan	Asia	1962	31.997	
Afghanistan	Asia	1967	34.020	
Afghanistan	Asia	1972	36.088	



mean\_life

59.47444

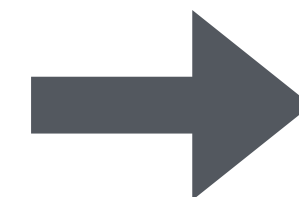
# summarise()

Compute table of summaries.

```
gapminder %>% summarise(mean_life = mean(lifeExp),  
                        min_life = min(lifeExp))
```

gapminder

country	continent	year	lifeExp	...
Afghanistan	Asia	1952	28.801	
Afghanistan	Asia	1957	30.332	
Afghanistan	Asia	1962	31.997	
Afghanistan	Asia	1967	34.020	
Afghanistan	Asia	1972	36.088	



mean_life	min_life
59.47444	23.599



# Summary functions

Take a vector as input, return a single value as output.

## Summary Functions

**TO USE WITH SUMMARISE ()**

**summarise()** applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function →

**COUNTS**

dplyr::n() - number of values/rows  
dplyr::n\_distinct() - # of uniques  
sum(!is.na()) - # of non-NA's

**LOCATION**

mean() - mean, also mean(is.na())  
median() - median

**LOGICALS**

mean() - Proportion of TRUE's  
sum() - # of TRUE's

**POSITION/ORDER**

dplyr::first() - first value  
dplyr::last() - last value  
dplyr::nth() - value in nth location of vector

**RANK**

quantile() - nth quantile  
min() - minimum value  
max() - maximum value

**SPREAD**

IQR() - Inter-Quartile Range  
mad() - mean absolute deviation  
sd() - standard deviation  
var() - variance

### Vectorized Functions

**TO USE WITH MUTATE ()**

**mutate()** and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

vectorized function →

**OFFSETS**

dplyr::lag() - Offset elements by 1  
dplyr::lead() - Offset elements by -1

**CUMULATIVE AGGREGATES**

dplyr::cumall() - Cumulative all()  
dplyr::cumany() - Cumulative any()  
cummax() - Cumulative max()  
dplyr::cummean() - Cumulative mean()  
cummin() - Cumulative min()  
cumprod() - Cumulative prod()  
cumsum() - Cumulative sum()

**RANKINGS**

dplyr::cumc\_dist() - Proportion of all values <= dplyr::dense\_rank() - rank with ties = min, no gaps  
dplyr::min\_rank() - rank with ties = min  
dplyr::ntile() - bins into n bins  
dplyr::percent\_rank() - min\_rank scaled to [0,1]  
dplyr::row\_number() - rank with ties = "first"

**MATH**

+ , -, \*, /, ^, %/%, %% - arithmetic ops  
log(), log2(), log10() - logs  
<, <=, >, >=, !=, == - logical comparisons

**MISC**

dplyr::between() - x >= left & x <= right  
dplyr::case\_when() - multi-case if\_else()  
dplyr::coalesce() - first non-NA values by element across a set of vectors  
dplyr::if\_else() - element-wise if() + else()  
dplyr::na\_if() - replace specific values with NA  
p\_max() - element-wise max()  
p\_min() - element-wise min()  
dplyr::recode() - Vectorized switch()  
dplyr::recode\_factor() - Vectorized switch() for factors

## Summary Functions

**TO USE WITH SUMMARISE ()**

**summarise()** applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function →

**COUNTS**

dplyr::n() - number of values/rows  
dplyr::n\_distinct() - # of unques  
sum(!is.na()) - # of non-NA's

**LOCATION**

mean() - mean, also mean(is.na())  
median() - median

**LOGICALS**

mean() - Proportion of TRUE's  
sum() - # of TRUE's

**POSITION/ORDER**

dplyr::first() - first value  
dplyr::last() - last value  
dplyr::nth() - value in nth location of vector

**RANK**

quantile() - nth quantile  
min() - minimum value  
max() - maximum value

**SPREAD**

IQR() - Inter-Quartile Range  
mad() - mean absolute deviation  
sd() - standard deviation  
var() - variance

## Combine Tables

**COMBINE VARIABLES**

Use **bind\_cols()** to paste tables beside each other as they are.

**bind\_cols(...)** Returns tables placed side by side as a single table.  
**BE SURE THAT ROWS ALIGN.**

Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

- left\_join(x, y, by = NULL, copy = FALSE, suffix=c("x","y"),...)**  
Join matching values from y to x.
- right\_join(x, y, by = NULL, copy = FALSE, suffix=c("x","y"),...)**  
Join matching values from x to y.
- inner\_join(x, y, by = NULL, copy = FALSE, suffix=c("x","y"),...)**  
Join data. Retain only rows with matches.
- full\_join(x, y, by = NULL, copy = FALSE, suffix=c("x","y"),...)**  
Join data. Retain all values, all rows.

**COMBINE CASES**

Use **bind\_rows()** to paste tables below each other as they are.

**bind\_rows(..., id = NULL)**  
Returns tables one on top of the other as a single table. Set id to a column name to add a column of the original table names (as pictured).

- intersect(x, y, ...)**  
Rows that appear in both x and z.
- setdiff(x, y, ...)**  
Rows that appear in x but not z.
- union(x, y, ...)**  
Rows that appear in x or z. (Duplicates removed). union\_all() retains duplicates.

Use a **separate()** to test whether two data sets contain the exact same rows (in any order).

## Row Names

Tidy data does not use rownames, which stores a variable outside of the columns. To work with the rownames, first move them into a column.

**rownames\_to\_column()**  
Move row names into col.  
a <- rownames\_to\_column(mtcars, var = "r")

**column\_to\_rownames()**  
Move col in row names.  
column\_to\_rownames(a, var = "C")

**Also has rownames(), remove\_rownames()**





# Your Turn 6

Use `summarise()` to compute three statistics *about the data*:

1. The first (minimum) year in the dataset
2. The last (maximum) year in the dataset
3. The number of countries represented in the data (Hint: use cheatsheet)

03:00

```
gapminder %>%  
  summarise(first = min(year),  
            last = max(year),  
            n_countries = n_distinct(country))
```

```
# A tibble: 1 x 3  
#   first last n_countries  
#   <dbl> <dbl>     <int>  
# 1  1952  2007         142
```





# Your Turn 7

Extract the rows where `continent == "Africa"` and `year == 2007`.

Then use `summarise()` and summary functions to find:

- The number of unique countries
- The median life expectancy

03:00

```
gapminder %>%  
  filter(continent == "Africa", year == 2007) %>%  
  summarise(n_countries = n_distinct(country), med_le = median(lifeExp))  
  
## A tibble: 1 x 2  
#   n_countries med_life_exp  
#       <int>       <dbl>  
#1         52       52.9265
```



# Grouping Cases

# group\_by()

Groups cases by common values of one or more columns.

```
gapminder %>%
```

```
  group_by(continent)
```

**In console**

```
# A tibble: 1,704 x 6
```

```
# Groups:   continent [5]
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fctr>	<fctr>	<int>	<dbl>	<int>	<dbl>
1	Afghanistan	Asia	1952	28.801	8425333	779.4453
2	Afghanistan	Asia	1957	30.332	9240934	820.8530
3	Afghanistan	Asia	1962	31.997	10267083	853.1007





# group\_by()

Groups cases by common values, then summarise acts by group

```
gapminder %>%  
  group_by(continent) %>%  
  summarise(n_countries = n_distinct(country))
```

continent	n_countries
Africa	52
Americas	25
Asia	33
Europe	30
Oceania	2



# Transform Data Notebook

```
01-manipulate-data.Rmd x
1 |---
2 title: "Manipulate Data"
3 output: html_document
4 ---
5
6 ```{r setup}
7 library(tidyverse)
8 library(babynames)
9
10 # Toy dataset to use
11 pollution <- tribble(
12   ~city, ~size, ~amount,
13   "New York", "large", 23,
14   "New York", "small", 14,
15   "London", "large", 22,
16   "London", "small", 16,
17   "Beijing", "large", 121,
18   "Beijing", "small", 56
19 )
20
21
22 ## babynames
23
24 ```{r}
25 babynames
```

```
pollution <- tribble(
  ~city, ~size, ~amount,
  "New York", "large", 23,
  "New York", "small", 14,
  "London", "large", 22,
  "London", "small", 16,
  "Beijing", "large", 121,
  "Beijing", "small", 56
)
```

Toy data set to  
practice with



```

pollution <- tribble(
  ~city, ~size, ~amount,
  "New York", "large", 23,
  "New York", "small", 14,
  "London", "large", 22,
  "London", "small", 16,
  "Beijing", "large", 121,
  "Beijing", "small", 56
)

```

pollution

city	particle size	amount ( $\mu\text{g}/\text{m}^3$ )
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



city	particle size	amount ( $\mu\text{g}/\text{m}^3$ )
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



mean	sum	n
42	252	6

```
pollution %>%
```

```
  summarise(mean = mean(amount), sum = sum(amount), n = n())
```

city	particle size	amount ( $\mu\text{g}/\text{m}^3$ )
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

mean	sum	n
42	252	6



city	particle size	amount ( $\mu\text{g}/\text{m}^3$ )
New York	large	23
New York	small	14



mean	sum	n
18.5	37	2

London	large	22
London	small	16



19.0	38	2
------	----	---

Beijing	large	121
Beijing	small	56

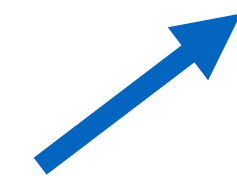


88.5	177	2
------	-----	---

`group_by() + summarise()`



city	particle size	amount ( $\mu\text{g}/\text{m}^3$ )
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



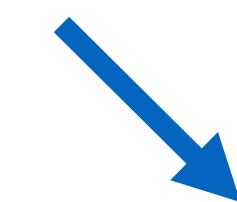
city	particle size	amount ( $\mu\text{g}/\text{m}^3$ )
New York	large	23
New York	small	14



London	large	22
London	small	16



Beijing	large	121
Beijing	small	56



city	mean	sum	n
New York	18.5	37	2
London	19.0	38	2
Beijing	88.5	177	2

```
pollution %>%
```

```
  group_by(city) %>%
```

```
  summarise(mean = mean(amount), sum = sum(amount), n = n())
```

# Your Turn 8

Find the median life expectancy by continent in 2007.

05:59

```
gapminder %>%  
  filter(year == 2007) %>%  
  group_by(continent) %>%  
  summarise(med_life_exp = median(lifeExp))
```

<b>continent</b> <fctr>	<b>med_life_exp</b> <dbl>
Africa	52.9265
Americas	72.8990
Asia	72.3960
Europe	78.6085
Oceania	80.7195

5 rows

# Challenge

# Task

I want to find the countries with **biggest jump** in life expectancy (between any two consecutive records).



# Your Turn 9

## Brainstorm with your neighbour

What sequence of operations would you need to do?

(**Hint:** `mutate()` will respect groups too)

I want to find the countries with **biggest jump** in life expectancy (between any two consecutive records).

05:00

# Your Turn 10

## Putting it all together

Find the country with biggest jump in life expectancy (between any two consecutive records).

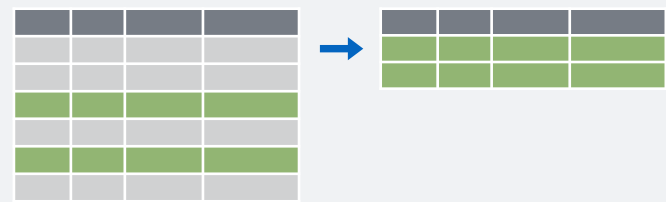
05:00



```
# One of many solutions
gapminder %>%
  group_by(country) %>%
  mutate(prev_lifeExp = lag(lifeExp),
         jump = lifeExp - prev_lifeExp) %>%
  arrange(desc(jump))
```



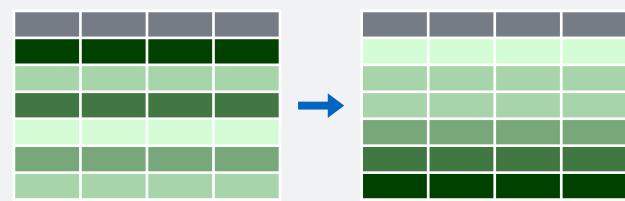
# dplyr: Data manipulation



Extract cases with **filter()**



Extract variables with **select()**



Arrange cases, with **arrange()**.



Make new variables, with **mutate()**.

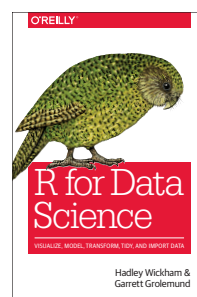


Make tables of summaries with **summarise()**.

along with **group\_by()**



# Joining datasets



In R4DS

**Relational Data**



# Joins

**Mutating joins** use information from one data set **to add variables** to another data set (like **mutate()**)

**Filtering joins** use information from one data set **to extract cases** from another data set (like **filter()**)



# Common Syntax

Each join function returns a data frame / tibble.

```
left_join(x, y, by = NULL, ... )
```

join function

data frames  
to join

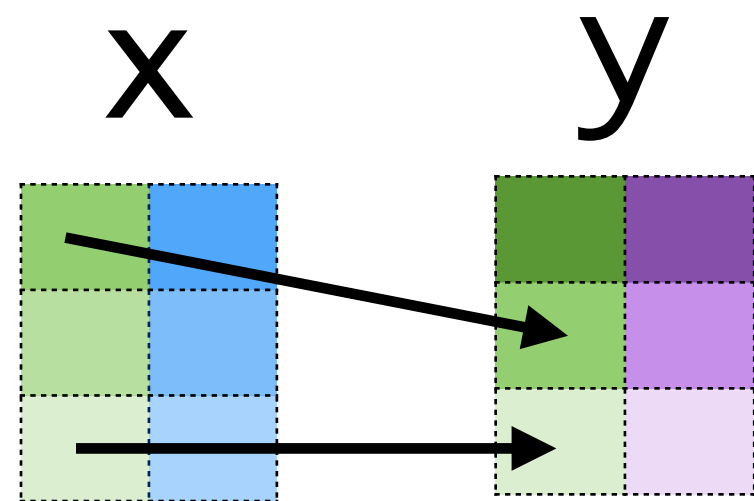
(optional) names of  
columns to join on

```
x %>% left_join(y, by = NULL, ... )
```

In pipe  
form



## Two table verbs



## Mutating joins

Columns from x and y

Green	Blue	Purple
Green	Blue	NA
Light Green	Light Blue	Light Purple

All rows in x

`x %>% left_join(y)`

Green	NA	Purple
Green	Blue	Purple
Light Green	Light Blue	Light Purple

All rows in y

`x %>% right_join(y)`

Green	Blue	Purple
Light Green	Light Blue	Light Purple

Only rows in x with matches in y

`x %>% inner_join(y)`

Green	Blue	Purple
Green	Blue	NA
Light Green	Light Blue	Light Purple
Green	NA	Purple

All rows from x and y

`x %>% full_join(y)`

## Filtering joins

Columns from x

Green	Blue
Light Green	Light Blue

Rows in x that have matches in y

`x %>% semi_join(y)`

Light Green	Light Blue
-------------	------------

Rows in x that don't have matches in y

`x %>% anti_join(y)`



DataCamp

www.datacamp.com

# Your Turn 11

Use `left_join()` to add the country codes in `country_codes` to the `gapminder` data.

**Challenge** Which codes in `country_codes` have no matches in `gapminder`?

02:00



```
gapminder %>% left_join(country_codes)
```

country <chr>	continent <fctr>	year <int>	lifeExp <dbl>	pop <int>	gdpPercap <dbl>	iso_alpha <chr>	
Afghanistan	Asia	1952	28.80100	8425333	779.4453	AFG	
Afghanistan	Asia	1957	30.33200	9240934	820.8530	AFG	
Afghanistan	Asia	1962	31.99700	10267083	853.1007	AFG	
Afghanistan	Asia	1967	34.02000	11537966	836.1971	AFG	
Afghanistan	Asia	1972	36.08800	13079460	739.9811	AFG	
Afghanistan	Asia	1977	38.43800	14880372	786.1134	AFG	
Afghanistan	Asia	1982	39.85400	12881816	978.0114	AFG	
Afghanistan	Asia	1987	40.82200	13867957	852.3959	AFG	
Afghanistan	Asia	1992	41.67400	16317921	649.3414	AFG	
Afghanistan	Asia	1997	41.76300	22227415	635.3414	AFG	

1-10 of 1,704 rows | 1-7 of 8 columns

Previous 1 2 3 4 5 6 ... 100 Next





# Tidy tools

# Tidy tools

Functions are easiest to use when they are:

1. **Simple** - They do one thing, and they do it well
2. **Composable** - They can be combined with other functions for multi-step operations
3. **Smart** - They can use R objects as input.

Tidy functions do these things in a specific way.



# 1. Simple

They do one thing, and they do it well

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

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

**group\_by()** - group **cases**

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

**mutate()** - create new **variables**

**summarise()** - summarise **variables** / create **cases**



## 2. Composable

They can be combined with other functions for multi-step operations

```
gapminder %>%  
  filter(year == 2007) %>%  
  arrange(desc(lifeExp))
```

Each dplyr function takes a tibble as its first argument and returns a tibble. As a result, you can directly pipe the output of one function into the next.



# 3. Smart

They can use R objects as input.

```
years <- 2001:2011  
gapminder %>%  
  filter(year %in% years)
```

Found in .data,  
no need for \$

Found in global  
environment





# Careful!

This doesn't work:

```
var <- "pop"  
gapminder %>%  
  mutate(mean_var = mean(var))
```

This does:

```
var <- quo(pop)  
gapminder %>%  
  mutate(mean_var = mean(!var))
```

See <http://dplyr.tidyverse.org/articles/programming.html>



# Transform Data with

