MONASH University
MALAYSIA

MONASH
BUSINESS

Data Manipulation with dplyr

ETW2001 Foundations of Data Analysis and Modelling

Manjeevan Singh Seera

Accredited by:

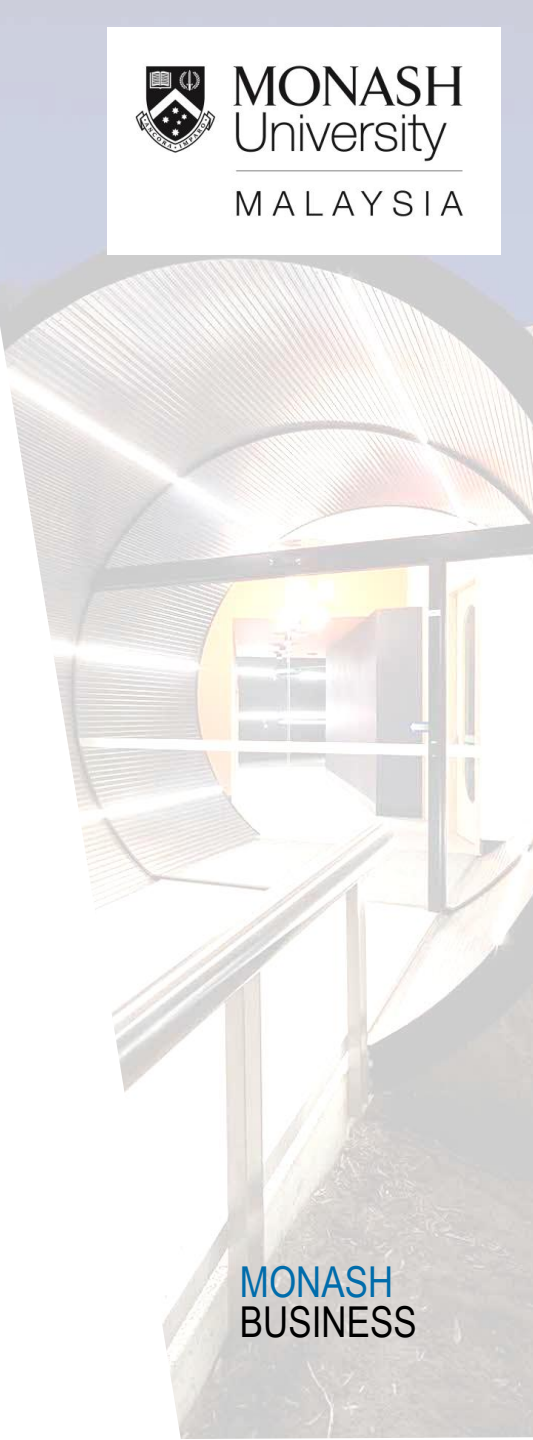
AACSB ACCREDITED

Advanced Signatory:

PRME
an initiative of the
United Nations Global Compact

Outline

- Data Manipulation with dplyr
  - Filter and Arrange Rows
  - Select Columns
  - Add New Variables
  - Grouped Summaries



MONASH University MALAYSIA

MONASH BUSINESS

Visualization

Visualization is an important tool for insight generation, but it is **rare** that you get the data in **exactly** the right form you need.

Often you'll need to create some new variables or summaries, or maybe you just want to rename the variables or reorder the observations in order to make the data a little easier to work with.

In the next slides, we will transform data using the dplyr package and a dataset on flights departing New York City in 2013.

Source: https://r4ds.had.co.nz/

Prerequisites

In the next slides, the focus is on using the dplyr package, another core member of the tidyverse. We'll illustrate the key ideas using data from the nycflights13 package, and use ggplot2 to help us understand the data.

There are a few conflicts when loading tidyverse. It tells you that dplyr overwrites some functions in base R. To use the base version of these functions after loading dplyr, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

```
library(tidyverse)
install.packages("nycflights13")
library(nycflights13)
```

```
── Attaching packages ─────────────────────
✓ ggplot2 3.3.2     ✓ purrr   0.3.4
✓ tibble  3.0.4     ✓ dplyr   1.0.2
✓ tidyr   1.1.2     ✓ stringr 1.4.0
✓ readr   1.4.0     ✓ forcats 0.5.0

── Conflicts ────────────────────────────
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

MONASH University MALAYSIA

MONASH BUSINESS

nycflights13

To explore the basic data manipulation in dplyr, lets use `nycflights13::flights`.

This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?flights`.

```
flights
```

A tibble: 336776 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00
2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183	1576	5	45	2013-01-01 05:00:00
2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116	762	6	0	2013-01-01 06:00:00
2013	1	1	554	558	-4	740	728	12	UA	1696	N39463	EWR	ORD	150	719	5	58	2013-01-01 05:00:00
2013	1	1	555	600	-5	913	854	19	B6	507	N516JB	EWR	FLL	158	1065	6	0	2013-01-01 06:00:00
2013	1	1	557	600	-3	709	723	-14	EV	5708	N829AS	LGA	IAD	53	229	6	0	2013-01-01 06:00:00
2013	1	1	557	600	-3	838	846	-8	B6	79	N593JB	JFK	MCO	140	944	6	0	2013-01-01 06:00:00
2013	1	1	558	600	-2	753	745	8	AA	301	N3ALAA	LGA	ORD	138	733	6	0	2013-01-01 06:00:00

nycflights13

You might notice that this data frame prints **a little differently** from other data frames you might have used in the past: it only shows the **first few rows** and all the **columns that fit** on one screen.

2013	1	1	615	615	0	1039	1100	-21	B6	709	N794JB	JFK	SJU	182	1598	6	15	2013-01-01 06:00:00
2013	1	1	615	615	0	833	842	-9	DL	575	N326NB	EWR	ATL	120	746	6	15	2013-01-01 06:00:00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2013	9	30	2123	2125	-2	2223	2247	-24	EV	5489	N712EV	LGA	CHO	45	305	21	25	2013-09-30 21:00:00
2013	9	30	2127	2129	-2	2314	2323	-9	EV	3833	N16546	EWR	CLT	72	529	21	29	2013-09-30 21:00:00

To see the whole dataset, you can run `view(flights)` which will open the dataset in R. It prints differently because it's a **tibble**.

- Tibbles are data frames, but slightly tweaked to work better in the tidyverse.

nycflights13

You might also have noticed the row of three (or four) letter abbreviations under the column names.

These describe the type of each variable:

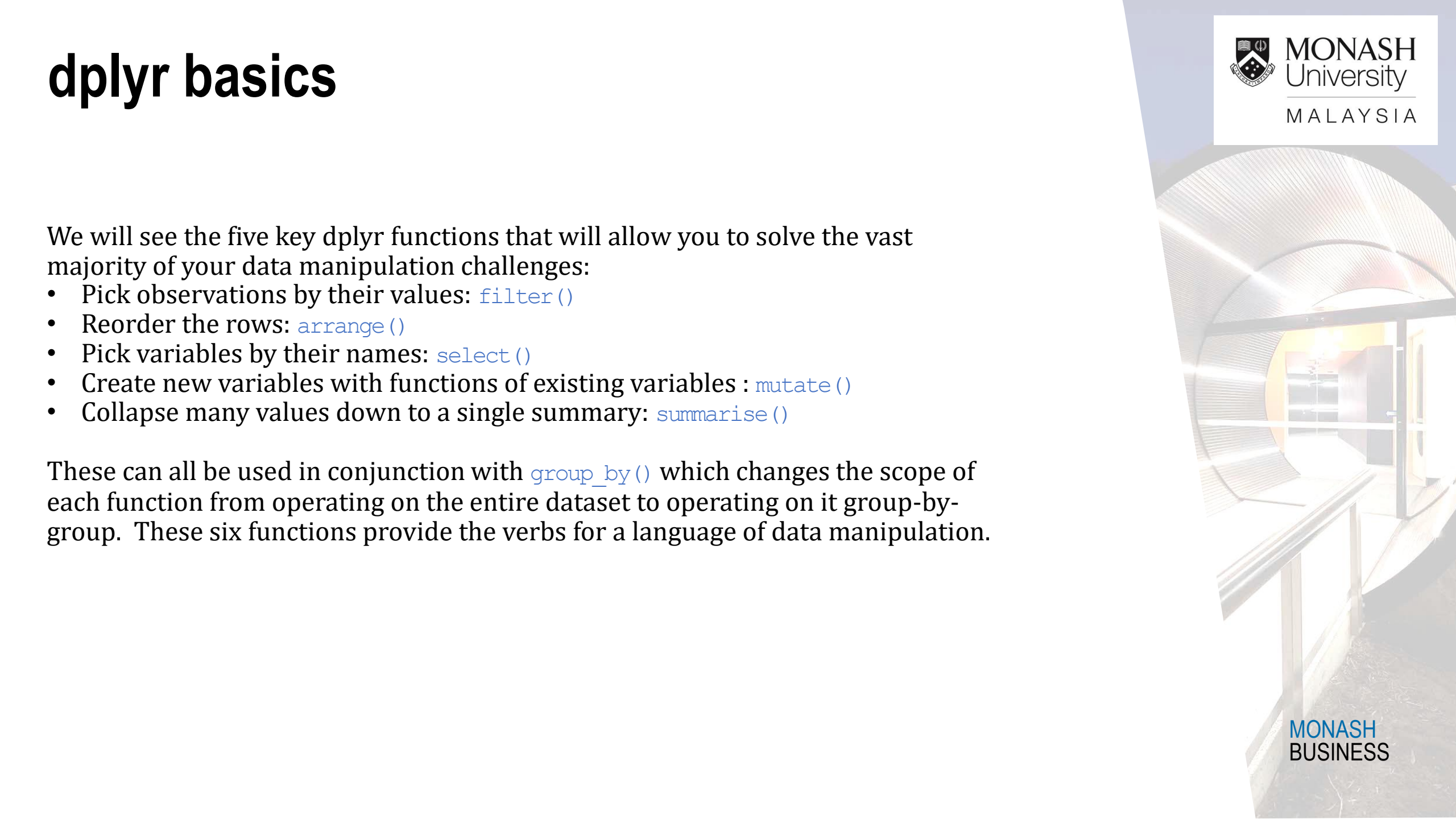
`Int`	integers,
`dbl`	doubles, or real numbers,
`chr`	character vectors, or strings,
`dttm`	date-times (a date + a time),
`lgl`	logical, vectors that contain only TRUE or FALSE,
`fctr`	factors, which R uses to represent categorical variables with fixed possible values.
`date`	dates.

dplyr basics

We will see the five key dplyr functions that will allow you to solve the vast majority of your data manipulation challenges:

- Pick observations by their values: `filter()`
- Reorder the rows: `arrange()`
- Pick variables by their names: `select()`
- Create new variables with functions of existing variables : `mutate()`
- Collapse many values down to a single summary: `summarise()`

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

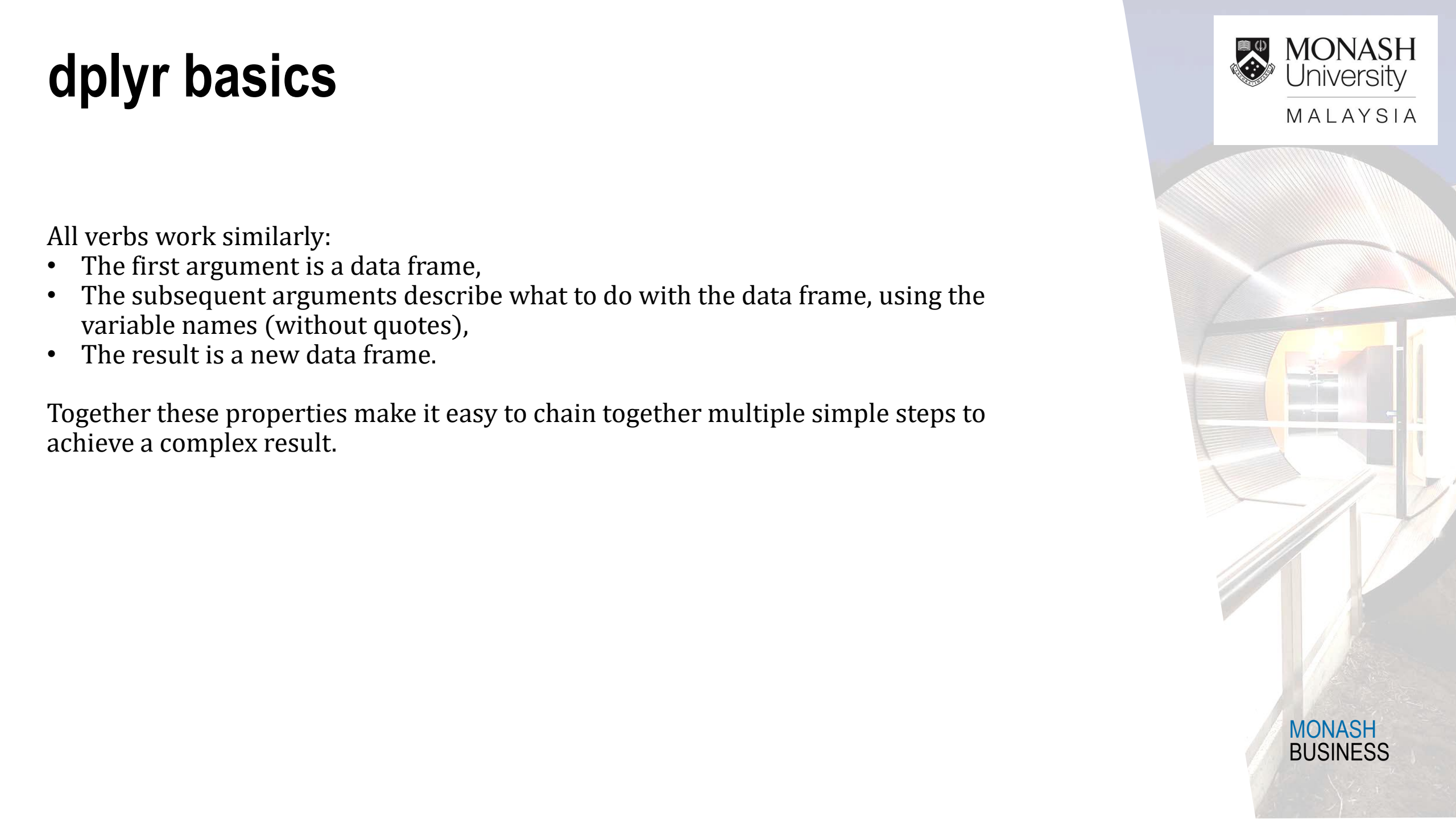


dplyr basics

All verbs work similarly:

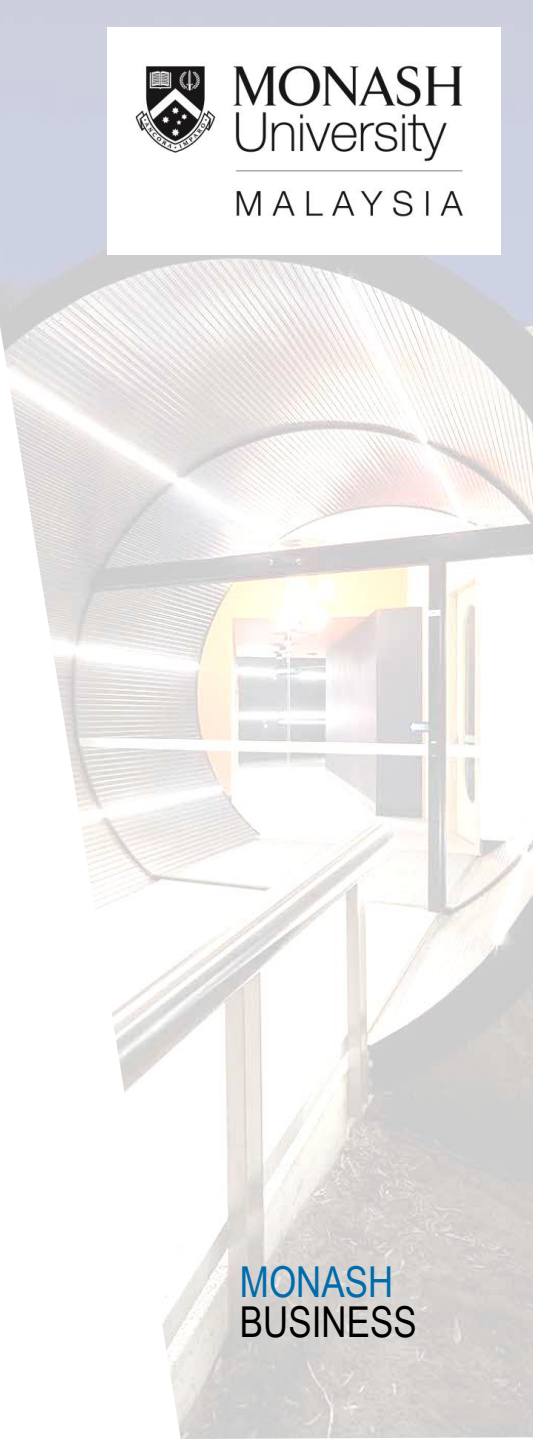
- The first argument is a data frame,
- The subsequent arguments describe what to do with the data frame, using the variable names (without quotes),
- The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.



Outline

- ✓ Data Manipulation with dplyr
  - ❑ **Filter and Arrange Rows**
  - ❑ Select Columns
  - ❑ Add New Variables
  - ❑ Grouped Summaries



MONASH University MALAYSIA

MONASH BUSINESS

Filter rows with filter()

`filter()` allows you to subset observations based on their values.

- First argument is the name of the data frame,
- Second and subsequent arguments are the expressions that filter the data frame.

For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
```

A tibble: 842 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00

Filter rows with filter()

When you run that line of code, dplyr executes the filtering operation and returns a new data frame.

dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, <–:

```
jan1 <- filter(flights, month == 1, day == 1)
jan1
```

A tibble: 842 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00

MONASH University MALAYSIA

MONASH BUSINESS

Filter rows with filter()

R either prints out the results, or saves them to a variable. Should you wish to do both, you can wrap the assignment in **parentheses**:

```
(dec25 <- filter(flights, month == 12, day == 25))
```

A tibble: 719 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	12	25	456	500	-4	649	651	-2	US	1895	N156UW	EWR	CLT	98	529	5	0	2013-12-25 05:00:00
2013	12	25	524	515	9	805	814	-9	UA	1016	N32404	EWR	IAH	203	1400	5	15	2013-12-25 05:00:00
2013	12	25	542	540	2	832	850	-18	AA	2243	N5EBAA	JFK	MIA	146	1089	5	40	2013-12-25 05:00:00

Comparisons

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

While being the most common operator, the most mistakes are made using `=` instead of `==` when testing for equality. When this happens you'll get an informative error:

```
filter(flights, month == 1)
```

A tibble: 27004 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00

Floating point

There's another common problem you might encounter when using == floating point numbers.

```
sqrt(2) ^ 2 == 2
```

FALSE

MONASH University MALAYSIA

MONASH BUSINESS

near

Computers use finite precision arithmetic, hence so remember that every number you see is an approximation.

Instead of relying on `==`, you can use `near()`:

```
near(sqrt(2) ^ 2,  2)
near(1 / 49 * 49, 1)
```

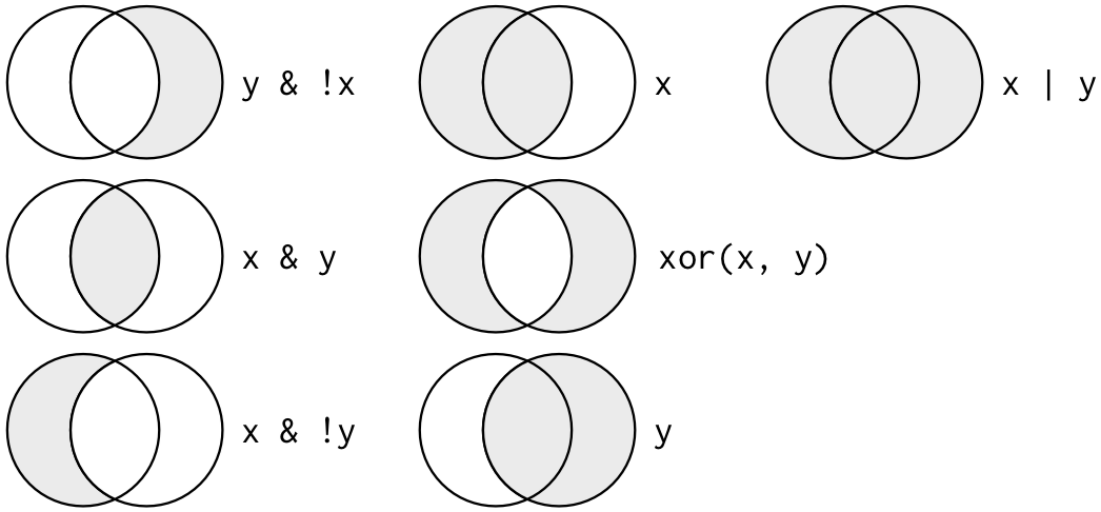
```
TRUE
TRUE
```

MONASH University MALAYSIA

MONASH BUSINESS

Logical operators

Multiple arguments to `filter()` are combined with "and": every expression must be true in order for a row to be included in the output. In other combinations, you'll need to use Boolean operators yourself: & is "and", | is "or", and ! is "not".



Complete set of Boolean operations. x is the left-hand circle, y is the right-hand circle, and the shaded region show which parts each operator selects.

Filter

Lets find flights that departed in November (11) or December (12):

```
filter(flights, month == 11 | month == 12)
```

A tibble: 55403 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	11	1	5	2359	6	352	345	7	B6	745	N568JB	JFK	PSE	205	1617	23	59	2013-11-01 23:00:00
2013	11	1	35	2250	105	123	2356	87	B6	1816	N353JB	JFK	SYR	36	209	22	50	2013-11-01 22:00:00
2013	11	1	455	500	-5	641	651	-10	US	1895	N192UW	EWR	CLT	88	529	5	0	2013-11-01 05:00:00

The codes don't work like **plain English**, such as `filter(flights, month == (11 | 12))`, which you might **literally translate** into "finds all flights that departed in November (11) or December (12)".

Instead it finds all months that equal `11 | 12`, an expression that evaluates to TRUE. In a numeric context (like here), TRUE becomes one (1), so this finds all flights in January (1), not November (11) or December (12).

MONASH University MALAYSIA

MONASH BUSINESS

Filter

A useful short-hand for this problem is `x %in% y`. This will select every row where x is one of the values in y. As with the previous slide, we can now rewrite the code as:

```
nov_dec <- filter(flights, month %in% c(11, 12))
nov_dec
```

A tibble: 55403 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	11	1	5	2359	6	352	345	7	B6	745	N568JB	JFK	PSE	205	1617	23	59	2013-11-01 23:00:00
2013	11	1	35	2250	105	123	2356	87	B6	1816	N353JB	JFK	SYR	36	209	22	50	2013-11-01 22:00:00
2013	11	1	455	500	-5	641	651	-10	US	1895	N192UW	EWR	CLT	88	529	5	0	2013-11-01 05:00:00

Filter

Say we want to find flights that weren't delayed (on arrival or departure) by more than two hours (120 mins), we could use either of the following two filters:

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

A tibble: 316050 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00

As well as & and |, R also has && and ||. Whenever you start using complicated, multipart expressions in `filter()`, consider making them explicit variables instead.

Arrange rows with arrange()

`arrange()` works similarly to `filter()` except that instead of selecting rows, it **changes their order**. It takes a data frame and a set of column names to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

```
arrange(flights, year, month, day)
```

A tibble: 336776 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00

desc

Use `desc()` to **re-order** by a column in descending order:

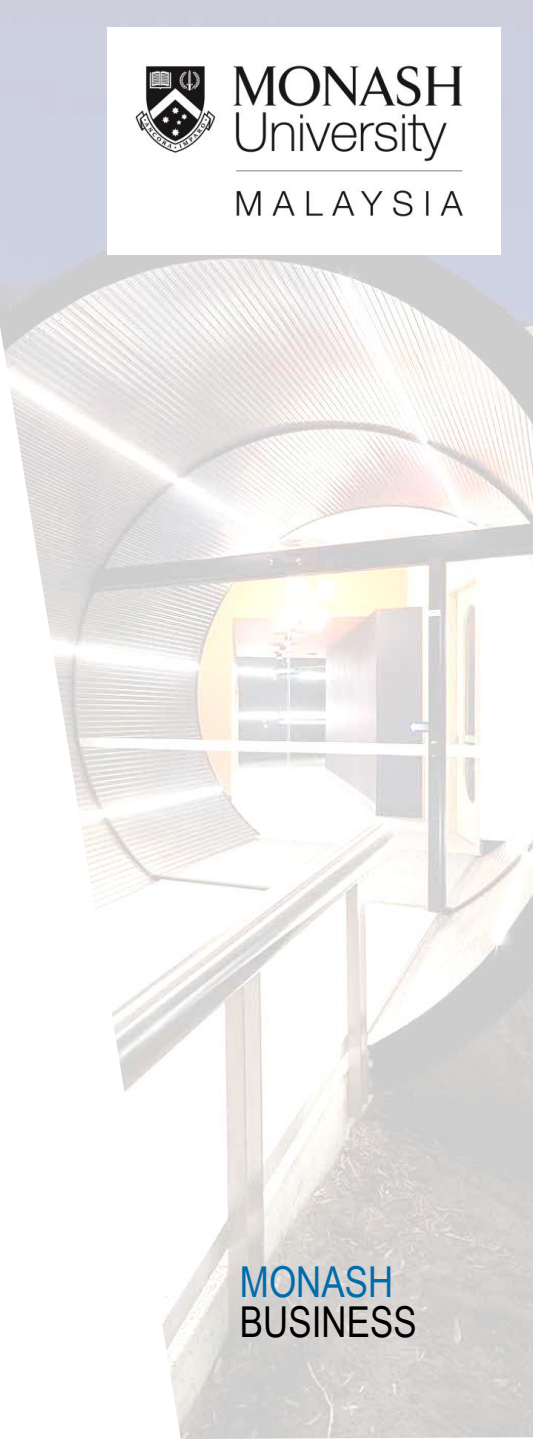
```
arrange(flights, desc(dep_delay))
```

A tibble: 336776 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	9	641	900	1301	1242	1530	1272	HA	51	N384HA	JFK	HNL	640	4983	9	0	2013-01-09 09:00:00
2013	6	15	1432	1935	1137	1607	2120	1127	MQ	3535	N504MQ	JFK	CMH	74	483	19	35	2013-06-15 19:00:00
2013	1	10	1121	1635	1126	1239	1810	1109	MQ	3695	N517MQ	EWR	ORD	111	719	16	35	2013-01-10 16:00:00

Outline

- ✓ Data Manipulation with dplyr
  - ✓ Filter and Arrange Rows
  - ❑ **Select Columns**
  - ❑ Add New Variables
  - ❑ Grouped Summaries



MONASH University MALAYSIA

MONASH BUSINESS

Select columns with select()

Huge datasets with thousands of variables are not uncommon. To narrow in on the variables you are interested in, `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

`select()` is not that helpful in the flights data as we have a mere 19 variables, but you can still get the general idea:

```
# Select columns by name
select(flights, year, month, day)
```

A tibble: 336776 × 3

year	month	day
<int>	<int>	<int>
2013	1	1
2013	1	1
2013	1	1

MONASH BUSINESS

Select columns with select()

There are a number of helper functions you can use within `select()`:

`starts_with("abc")`: matches names that begin with “abc”.

`ends_with("xyz")`: matches names that end with “xyz”.

`contains("ijk")`: matches names that contain “ijk”.

`matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters.

`num_range("x", 1:3)`: matches x1, x2 and x3.

Select columns with select()

`select()` can be used to rename variables, but it is not recommended as it drops all of the variables not explicitly mentioned.

Instead, use `rename()`, which is a variant of `select()` that keeps all the variables that aren't explicitly mentioned. In the example, the variable tailnum is renamed as tail_num.

```
rename(flights, tail_num = tailnum)
```

A tibble: 336776 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tail_num	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00

Select columns with select()

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame.

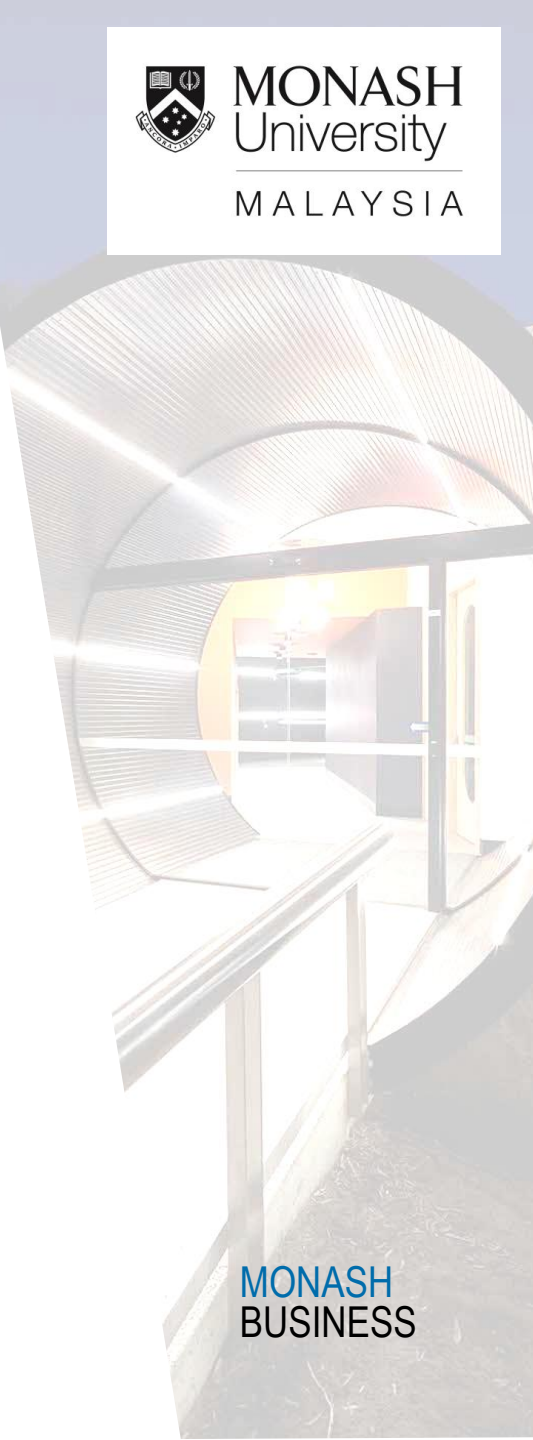
```
select(flights, time_hour, air_time, everything())
```

A tibble: 336776 × 19

time_hour	air_time	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	distance	hour	minute
<dttm>	<dbl>	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>
2013-01-01 05:00:00	227	2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	1400	5	15
2013-01-01 05:00:00	227	2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	1416	5	29
2013-01-01 05:00:00	160	2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	1089	5	40

Outline

- ✓ Data Manipulation with dplyr
  - ✓ Filter and Arrange Rows
  - ✓ Select Columns
  - ❑ **Add New Variables**
  - ❑ Grouped Summaries



MONASH University MALAYSIA

MONASH BUSINESS

Add new variables with mutate()

While we have learned on selecting existing columns, we may sometimes need to add new columns, which are functions of existing columns. This is the function of `mutate()`, which adds **new columns** at the end of the dataset.

Let's go back to the flights dataset. We will add 2 variables: gain and speed.

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60
)
```

A tibble: 336776 × 9

year	month	day	dep_delay	arr_delay	distance	air_time	gain	speed
<int>	<int>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
2013	1	1	2	11	1400	227	-9	370.0441
2013	1	1	4	20	1416	227	-16	374.2731
2013	1	1	2	33	1089	160	-31	408.3750

Add new variables with mutate()

We add a third variable, gain_per_hour from the 2 newly created variables.

```
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

A tibble: 336776 × 10

year	month	day	dep_delay	arr_delay	distance	air_time	gain	hours	gain_per_hour
<int>	<int>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
2013	1	1	2	11	1400	227	-9	3.7833333	-2.3788546
2013	1	1	4	20	1416	227	-16	3.7833333	-4.2290749
2013	1	1	2	33	1089	160	-31	2.6666667	-11.6250000

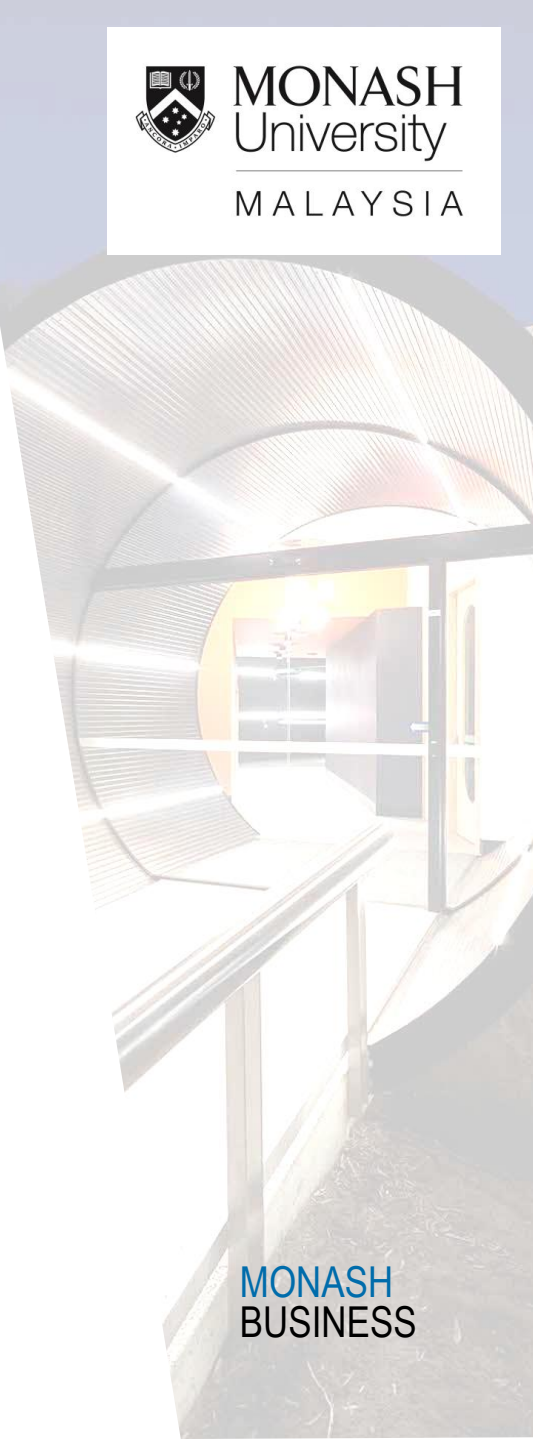
transmute

If you only want to keep the new variables, use `transmute()`:

```
transmute(flights,
  gain = dep_delay - arr_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

A tibble: 336776 × 3

gain	hours	gain_per_hour
<dbl>	<dbl>	<dbl>
-9	3.7833333	-2.3788546
-16	3.7833333	-4.2290749
-31	2.6666667	-11.6250000



MONASH University MALAYSIA

MONASH BUSINESS

Creation functions

There are various functions in creating variables with `mutate()`. Most importantly, it must be vectorised: it must take a vector of values as input, return a vector with the same number of values as output.

Here is a list to most functions you might use:

- Arithmetic operators: +, -, *, /, ^ (all vectorised). If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: air_time / 60, hours * 60 + minute, etc.
- Arithmetic operators in conjunction with the aggregate functions. For example, x / sum(x) calculates the proportion of a total, and y - mean(y) computes the difference from the mean.

Creation functions

Modular arithmetic: `%/%` (integer division) and `%%` (remainder), where `x == y * (x %/% y) + (x %% y)`. It is handy as it allows you to break integers up into pieces. As an example, you can compute hour and minute from `dep_time` with:

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)
```

A tibble: 336776 × 3

dep_time	hour	minute
<int>	<dbl>	<dbl>
517	5	17
533	5	33
542	5	42

MONASH University MALAYSIA

MONASH BUSINESS

Logs

Logs: `log()`, `log2()`, `log10()`. They are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude.

It is recommended to use `log2()` as it's easy to interpret: a difference of 1 on the log scale corresponds to doubling on the original scale and a difference of -1 corresponds to halving.

Offsets

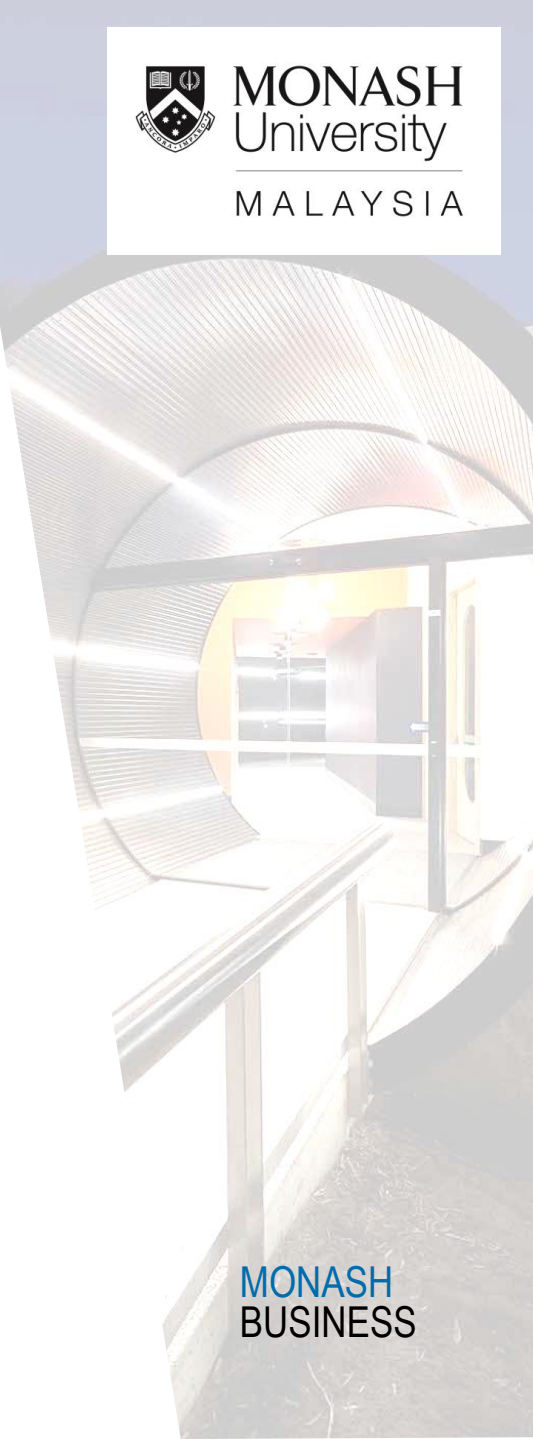
Offsets: `lead()` and `lag()` allow you to refer to leading or lagging values. This allows you to compute running differences (e.g. `x - lag(x)`) or find when values change `(x != lag(x))`. They are most useful in conjunction with `group_by()`.

```
x <- 1:10
lag(x)
```

<NA> · 1 · 2 · 3 · 4 · 5 · 6 · 7 · 8 · 9

```
lead(x)
```

2 · 3 · 4 · 5 · 6 · 7 · 8 · 9 · 10 · <NA>



MONASH University MALAYSIA

MONASH BUSINESS

Aggregates

Cumulative and rolling aggregates:

- R provides functions for running sums, products, mins and maxes: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`;
- dplyr provides `cummean()` for cumulative means.

```
cumsum(x)
```

1 · 3 · 6 · 10 · 15 · 21 · 28 · 36 · 45 · 55

```
cummean(x)
```

1 · 1.5 · 2 · 2.5 · 3 · 3.5 · 4 · 4.5 · 5 · 5.5

Logical comparisons

Logical comparisons consists of: <, <=, >, >=, !=, and ==.

If you're doing a complex sequence of logical operations it's often a good idea to store the interim values in new variables so you can check that each step is working as expected.

Ranking

Ranking: there are a number of ranking functions, but you should start with `min_rank()`. It does the most usual type of ranking (e.g. 1st, 2nd, 3rd, 4th).

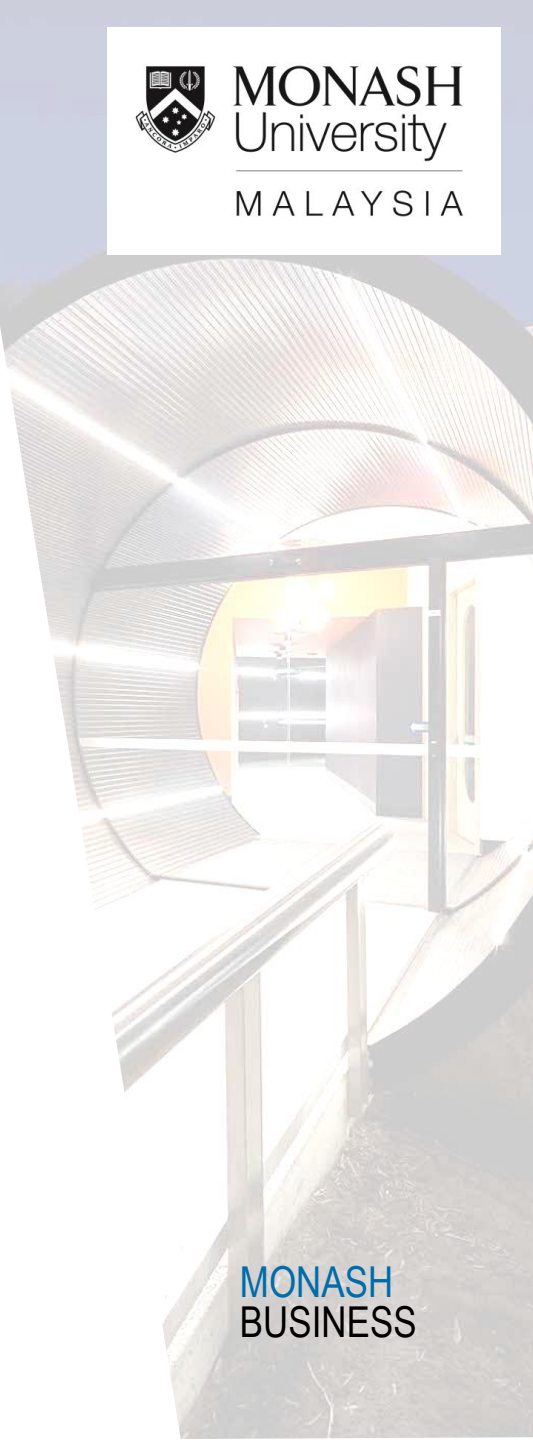
The default gives smallest values the small ranks; use `desc(x)` to give the largest values the smallest ranks.

```
y <- c(1, 2, 2, 3, 4, 8)
min_rank(y)
```

```
1 2 2 4 5 6
```

```
min_rank(desc(y))
```

```
6 4 4 3 2 1
```

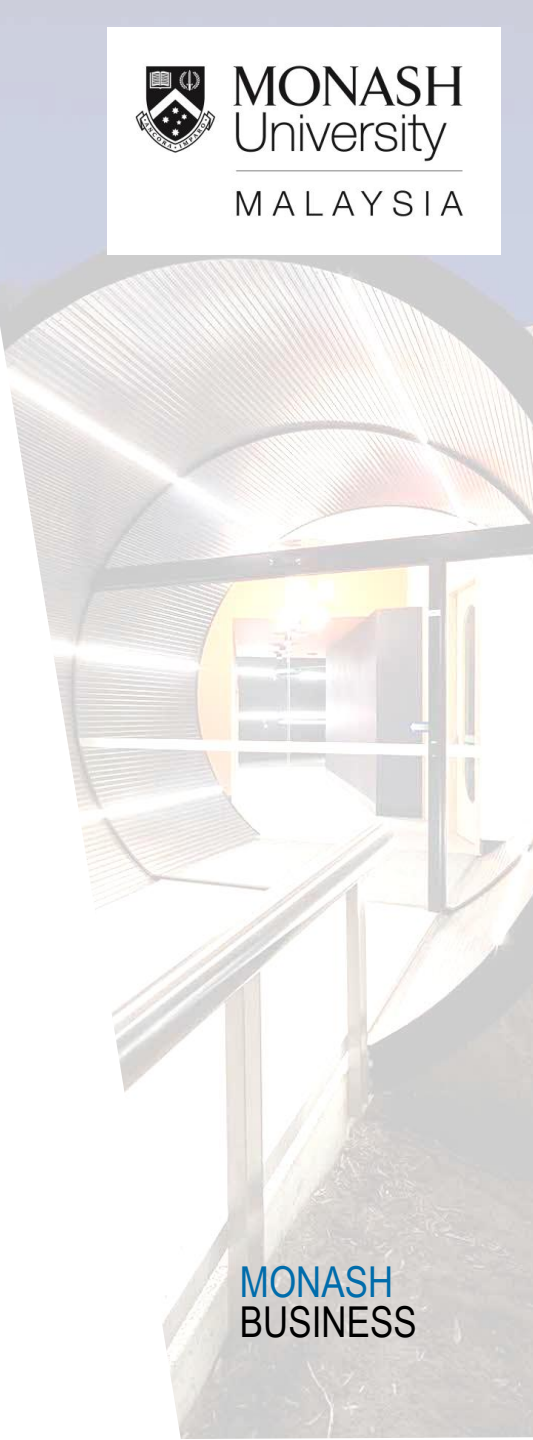


MONASH University MALAYSIA

MONASH BUSINESS

Outline

- ✓ Data Manipulation with dplyr
  - ✓ Filter and Arrange Rows
  - ✓ Select Columns
  - ✓ Add New Variables
  - ❑ **Grouped Summaries**



MONASH University MALAYSIA

MONASH BUSINESS

Grouped summaries with summarise()

The last key verb is `summarise()`. It collapses a data frame to a single row:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

A tibble: 1
× 1

delay
<dbl>
12.63907

MONASH University MALAYSIA

MONASH BUSINESS

Grouped summaries with summarise()

`summarise()` is that useful unless we use it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. When you use the dplyr verbs on a grouped data frame they'll be automatically applied "by group".

If we applied exactly the same code to a data frame grouped by date, we get the average delay per date:

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

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

A grouped_df: 365 × 4

year	month	day	delay
<int>	<int>	<int>	<dbl>
2013	1	1	11.548926
2013	1	2	13.858824
2013	1	3	10.987832

Grouped summaries with summarise()

Together `group_by()` and `summarise()` provide one of the tools that you'll use most commonly when working with dplyr: **grouped summaries**.

But before we go any further with this, we need to introduce a powerful new idea: **the pipe**.

MONASH University MALAYSIA

MONASH BUSINESS

Combining multiple operations using pipe

Say that we want to explore the relationship between the **distance** and **average** delay for each location. Using what you know about dplyr, you can write the following code:

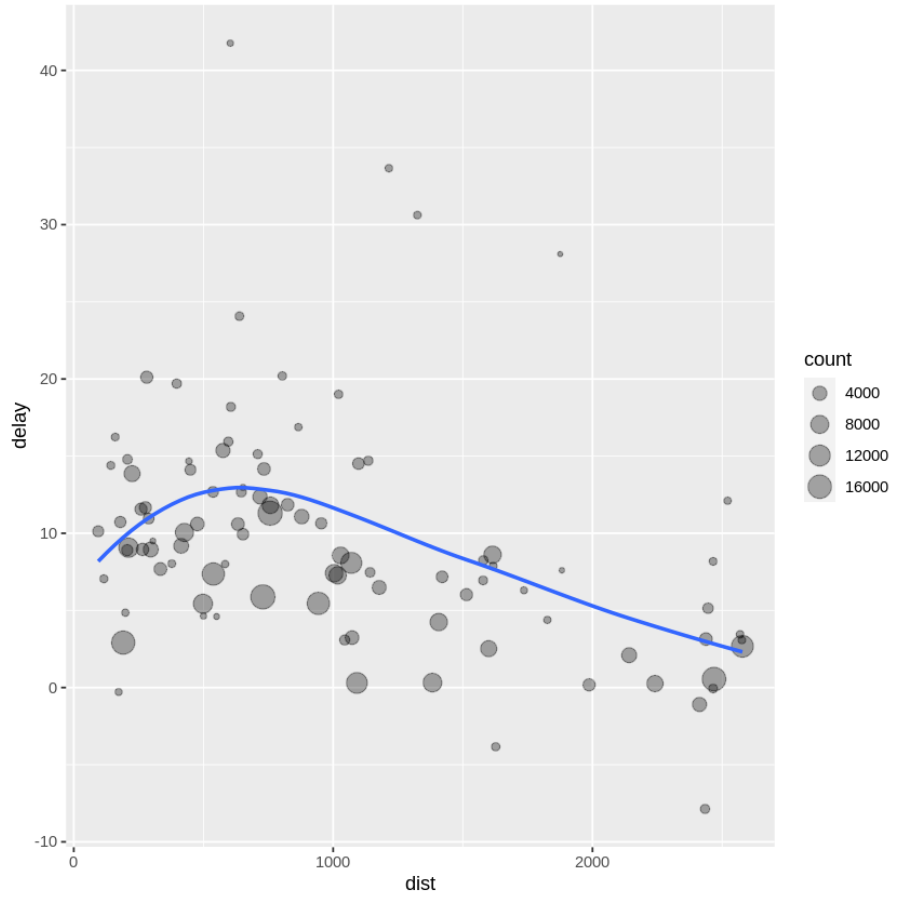
```
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)
```

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

```
delay <- filter(delay, count > 20, dest != "HNL")

ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```

```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



MONASH University MALAYSIA

MONASH BUSINESS

Combining multiple operations using pipe

There are three steps to prepare this data:

1. Group flights by destination,
2. Summarise to compute distance, average delay, and number of flights,
3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport.

There's another way to tackle the same problem with the pipe, %>%:

```
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

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

MONASH University MALAYSIA

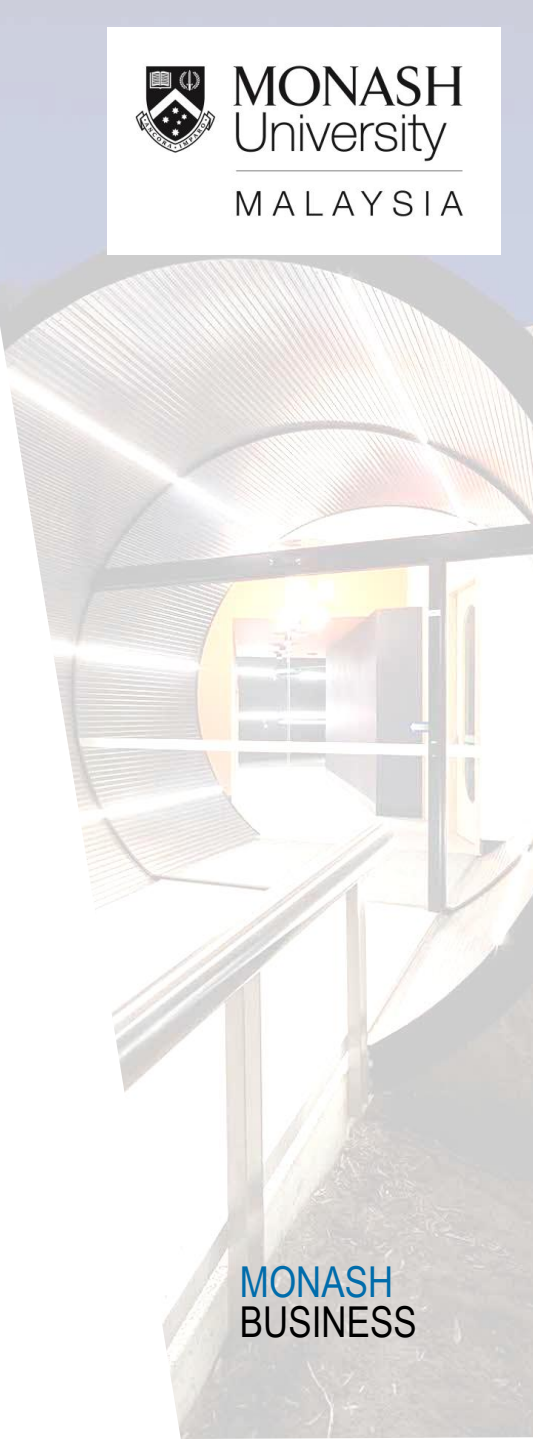
MONASH BUSINESS

Combining multiple operations using pipe

This can be read as a series of imperative statements: group, then summarise, then filter. A good way to pronounce %>% when reading code is "then".

Behind the scenes, x %>% f(y) turns into f(x, y), and x %>% f(y) %>% g(z) turns into g(f(x, y), z) and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom.

Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is ggplot2: it was written **before** the pipe was discovered.



MONASH University MALAYSIA

MONASH BUSINESS

Combining multiple operations using pipe

In this case, where missing values represent cancelled flights, we could also tackle the problem by first removing the cancelled flights. Let's save this as `not_cancelled` so that we can reuse this.

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

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

A grouped_df: 365 × 4

year	month	day	mean
<int>	<int>	<int>	<dbl>
2013	1	1	11.435620
2013	1	2	13.677802
2013	1	3	10.907778

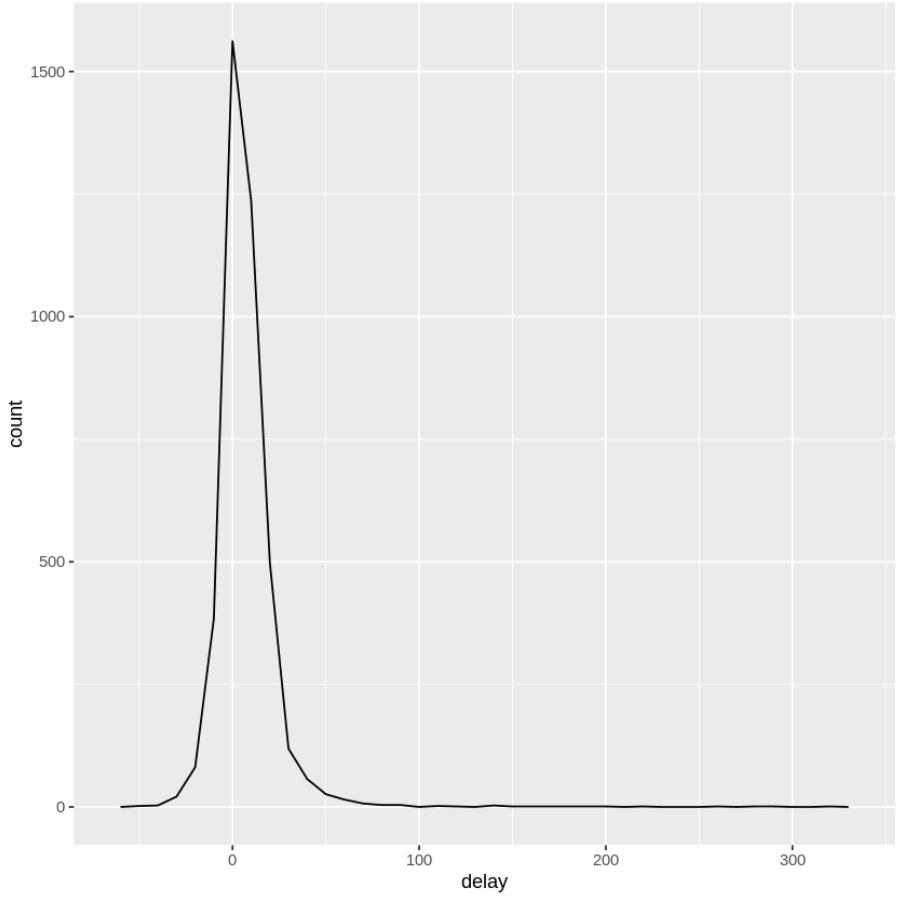
Counts

In any **aggregation**, it's good to include either a count (`n()`), or a count of non-missing values (`sum(!is.na(x))`). With that, you can check that you're not drawing conclusions based on very small amounts of data. As an example, let's look at the planes (identified by their tail number) that have the **highest average delays**:

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay)
  )
```

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

```
ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```



MONASH University MALAYSIA

MONASH BUSINESS

Counts

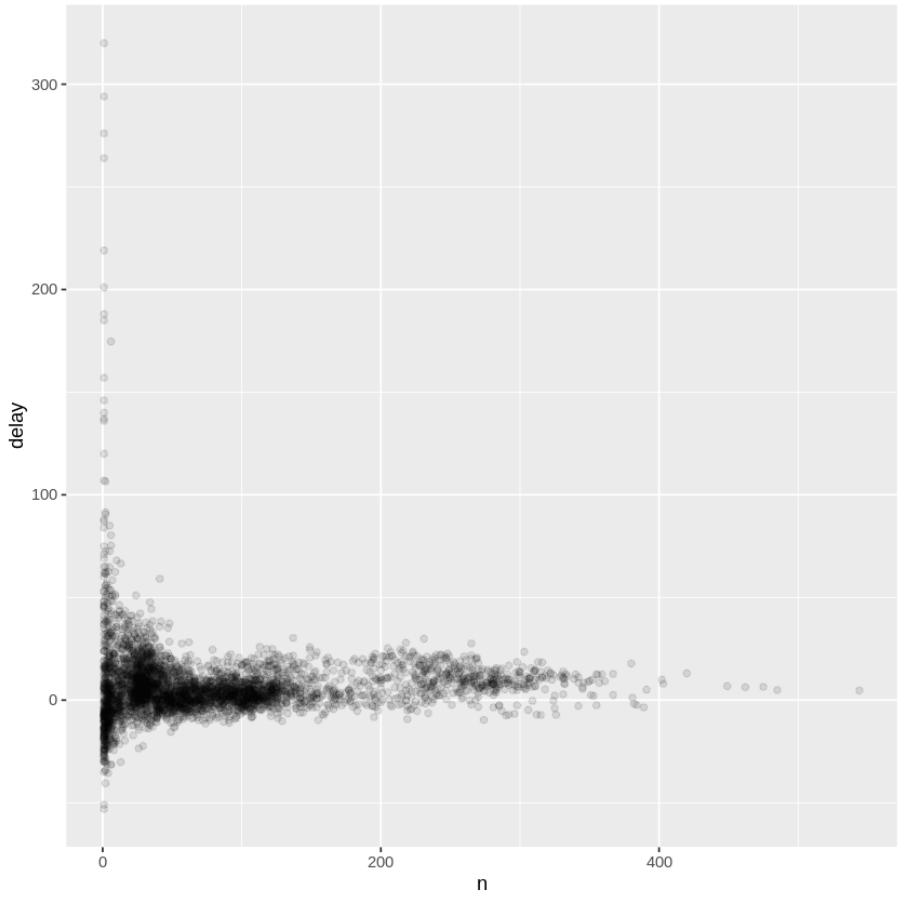
Some planes have an average delay of **5 hours** (300 minutes)!

We can get more insight if we draw a **scatterplot** of number of flights *vs*. average delay:

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
```

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

```
ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



MONASH University MALAYSIA

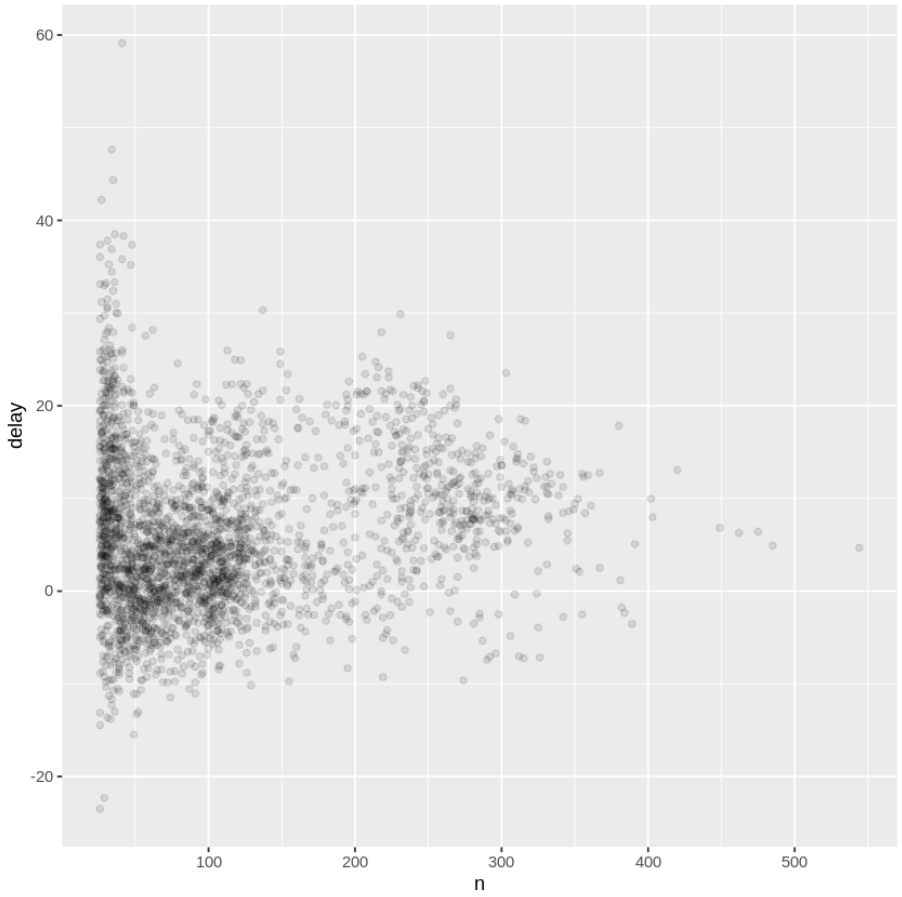
MONASH BUSINESS

Filter

It's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups.

This is what the following code does, as well as showing you a handy pattern for integrating ggplot2 into dplyr flows.

```
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
    geom_point(alpha = 1/10)
```



Useful summary functions

While using means, counts, and sum can get you a long way, there are many other useful summary functions.

Measures of location: we've used `mean(x)`, but `median(x)` is also useful.

- Mean is the sum divided by the length,
- Median is a value where 50% of x is above it, and 50% is below it.

Useful summary functions

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average positive delay
  )
```

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

A grouped_df: 365 × 5

year	month	day	avg_delay1	avg_delay2
<int>	<int>	<int>	<dbl>	<dbl>
2013	1	1	12.6510229	32.48156
2013	1	2	12.6928879	32.02991
2013	1	3	5.7333333	27.66087
2013	1	4	-1.9328194	28.30976
2013	1	5	-1.5258020	22.55882
2013	1	6	4.2364294	24.37270
2013	1	7	-4.9473118	27.76132

Spread

Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`. Root mean squared deviation, or standard deviation `sd(x)`, is the standard measure of spread. The interquartile range `IQR(x)` and median absolute deviation `mad(x)` are robust equivalents that may be more useful if you have outliers.

```
# Why is distance to some destinations more variable than to others?
not_cancelled %>%
  group_by(dest) %>%
  summarise(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))
```

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

A tibble: 104 × 2

dest	distance_sd
<chr>	<dbl>
EGE	10.542765
SAN	10.350094
SFO	10.216017

MONASH BUSINESS

Rank

Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Quantiles are a generalization of the median. As an example, `quantile(x, 0.25)` will find a value of x that is greater than 25% of the values, and less than the remaining 75%.

```
# When do the first and last flights leave each day?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = min(dep_time),
    last = max(dep_time)
  )
```

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

A grouped_df: 365 × 5

year	month	day	first	last
<int>	<int>	<int>	<int>	<int>
2013	1	1	517	2356
2013	1	2	42	2354
2013	1	3	32	2349

Position

Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`. These work similarly to x[1], x[2], and x[length(x)] but let you set a default value if that position does not exist. As an example, we can find the first and last departure for each day:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )
```

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

A grouped_df: 365 × 5

year	month	day	first_dep	last_dep
<int>	<int>	<int>	<int>	<int>
2013	1	1	517	2356
2013	1	2	42	2354
2013	1	3	32	2349

Filter

These functions are complementary to filtering on ranks. Filtering gives you all variables, with each observation in a separate row.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  mutate(r = min_rank(desc(dep_time))) %>%
  filter(r %in% range(r))
```

A grouped_df: 770 × 20

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour	r
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>	<int>
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00	831
2013	1	1	2356	2359	-3	425	437	-12	B6	727	N588JB	JFK	BQN	186	1576	23	59	2013-01-01 23:00:00	1
2013	1	2	42	2359	43	518	442	36	B6	707	N580JB	JFK	SJU	189	1598	23	59	2013-01-02 23:00:00	928

Counts

To **count** the number of non-missing values, use `sum(!is.na(x))`. To **count** the number of distinct (unique) values, use `n_distinct(x)`.

```
# Which destinations have the most carriers?
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
```

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

A tibble: 104 × 2

dest	carriers
<chr>	<int>
ATL	7
BOS	7
CLT	7

MONASH University MALAYSIA

MONASH BUSINESS

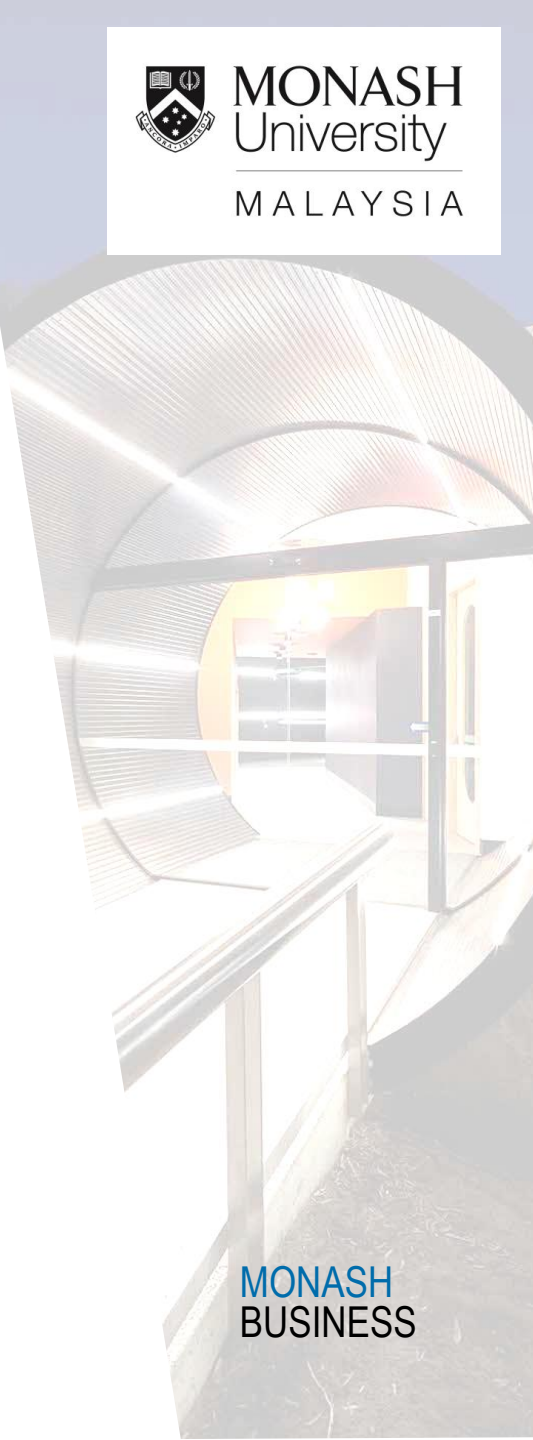
Counts

Counts are so useful that dplyr provides a simple helper:

```
not_cancelled %>%
  count(dest)
```

A tibble: 104 × 2

dest	n
<chr>	<int>
ABQ	254
ACK	264
ALB	418



MONASH University MALAYSIA

MONASH BUSINESS

Counts

You can optionally provide a **weight** variable. As an example, you could use this to "**count**" (sum) the total number of miles a plane flew:

```
not_cancelled %>%
  count(tailnum, wt = distance)
```

A tibble: 4037 × 2

tailnum	n
<chr>	<dbl>
D942DN	3418
N0EGMQ	239143
N10156	109664

MONASH University MALAYSIA

MONASH BUSINESS

Counts

Counts and proportions of logical values: `sum(x > 10)`, `mean(y == 0)`. When used with numeric functions, TRUE is converted to 1 and FALSE to 0. `sum(x)` gives the number of TRUEs in x, and `mean(x)` gives the proportion.

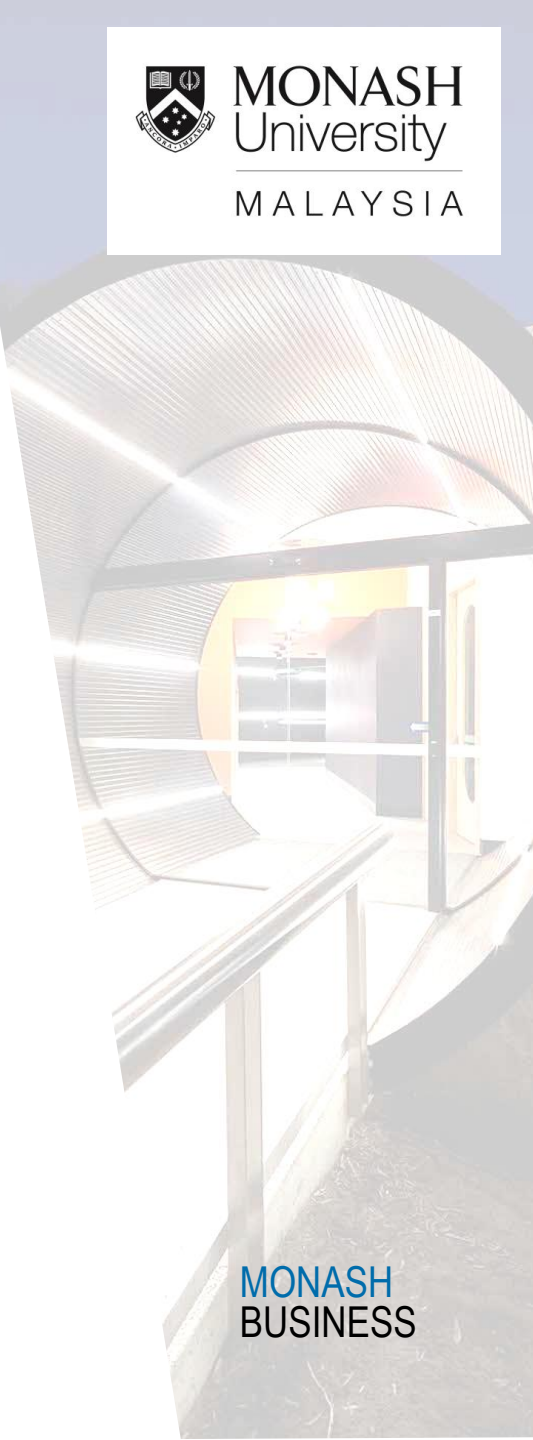
```
# How many flights left before 5am? (these usually indicate delayed
# flights from the previous day)
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))

# What proportion of flights are delayed by more than an hour?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(hour_prop = mean(arr_delay > 60))
```

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

A grouped_df: 365 × 4

year	month	day	n_early
<int>	<int>	<int>	<int>
2013	1	1	0
2013	1	2	3
2013	1	3	4



MONASH University MALAYSIA

MONASH BUSINESS

Grouping by multiple variables

When you group by multiple variables, each summary peels off one level of the grouping. This makes it simple to roll up a dataset.

```
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
```

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

A grouped_df: 365 × 4

year	month	day	flights
<int>	<int>	<int>	<int>
2013	1	1	842
2013	1	2	943
2013	1	3	914

Grouping by multiple variables

```
(per_month <- summarise(per_day, flights = sum(flights)))
```

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

A grouped_df: 12 × 3

year	month	flights
<int>	<int>	<int>
2013	1	27004
2013	2	24951
2013	3	28834

```
(per_year  <- summarise(per_month, flights = sum(flights)))
```

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

A tibble: 1 × 2

year	flights
<int>	<int>
2013	336776

Ungrouping

If you need to remove grouping, and return to operations on ungrouped data, use `ungroup()`.

```
daily %>%
  ungroup() %>%             # no longer grouped by date
  summarise(flights = n())  # all flights
```

A tibble:
1 × 1

flights
<int>
336776

MONASH University MALAYSIA

MONASH BUSINESS

THANK YOU

FIND OUT MORE AT MONASH.EDU.MY
LIKE @MONASH UNIVERSITY MALAYSIA ON FACEBOOK
FOLLOW @MONASHMALAYSIA ON TWITTER