Eddie Aguilar

Add new variables with mutate()

A data set often contains information that you can use to compute new variables. `mutate()` helps you compute those variables. Since `mutate()` always adds new columns to the end of a dataset, we'll start by creating a narrow dataset which will let us see the new variables (If we added new variables to `flights`, the new columns would run off the side of your screen, which would make them hard to see).

✓ select()

You can select a subset of variables by name with the `select()` function in dplyr. Run the code below to see the narrow data set that `select()` creates.

R Code | Start Over | Run Code

```
(flights_sml <- select(flights,
  arr_delay,
  dep_delay,
  distance,
  air_time
))
```

arr_delay	dep_delay	distance	air_time
<dbl>	<dbl>	<dbl>	<dbl>
11	2	1400	227
20	4	1416	227
33	2	1089	160
-18	-1	1576	183
-25	-6	762	116
12	-4	719	150
19	-5	1065	158
-14	-3	229	53
-8	-3	944	140
8	-2	733	138

1-10 of 1,000 rows Previous 1 2 3 4 5 6 ... 100 Next

✓ mutate()

The code below creates two new variables with dplyr's `mutate()` function. `mutate()` returns a new data frame that contains the new variables appended to a copy of the original data set. Take a moment to imagine what this will look like, and then click "Run Code" to find out.

R Code | Start Over | Run Code

```
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60
)
```

arr_delay	dep_delay	distance	air_time	gain	speed
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
11	2	1400	227	9	370.0441

Note that when you use `mutate()` you can create multiple variables at once, and you can even refer to variables that are created earlier in the call to create other variables later in the call:

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

arr_delay <dbl>	dep_delay <dbl>	distance <dbl>	air_time <dbl>	gain <dbl>	hours <dbl>
11	2	1400	227	9	3.7833333
20	4	1416	227	16	3.7833333
33	2	1089	160	31	2.6666667
-18	-1	1576	183	-17	3.0500000
-25	-6	762	116	-19	1.9333333
12	-4	719	150	16	2.5000000
19	-5	1065	158	24	2.6333333
-14	-3	229	53	-11	0.8833333
-8	-3	944	140	-5	2.3333333
8	-2	733	138	10	2.3000000

1-10 of 1,000 rows | 1-6 of 7 columns   Previous 1 2 3 4 5 6 ... 100 Next

✓ transmute()

`mutate()` will always return the new variables appended to a copy of the original data. If you want to return only the new variables, use `transmute()`. In the code below, replace `mutate()` with `transmute()` and then spot the difference in the results.

R Code   Start Over   Solution   Run Code   Submit Answer

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

gain <dbl>	hours <dbl>	gain_per_hour <dbl>
9	3.7833333	2.3788546
16	3.7833333	4.2290749
31	2.6666667	11.6250000
-17	3.0500000	-5.5737705
-19	1.9333333	-9.8275862
16	2.5000000	6.4000000
24	2.6333333	9.1139241
-11	0.8833333	-12.4528302

Useful mutate functions

You can use any function inside of `mutate()` so long as the function is **vectorised**. A vectorised function takes a vector of values as input and returns a vector with the same number of values as output.

Over time, I've found that several families of vectorised functions are particularly useful with `mutate()`:

- **Arithmetic operators**: `+`, `-`, `*`, `/`, `^`. These are all vectorised, using the so called "recycling rules". If one parameter is shorter than the other, it will automatically be repeated multiple times to create a vector of the same length. This is most useful when one of the arguments is a single number: `air_time / 60`, `hours * 60 + minute`, etc.

- **Modular arithmetic**: `%/%` (integer division) and `%%` (remainder), where `x == y * (x %/% y) + (x %% y)`. Modular arithmetic is a handy tool because it allows you to break integers up into pieces. For example, in the flights dataset, you can compute `hour` and `minute` from `dep_time` with:

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

dep_time	hour	minute
<int>	<dbl>	<dbl>
517	5	17
533	5	33
542	5	42
544	5	44
554	5	54
554	5	54
555	5	55
557	5	57
557	5	57
558	5	58

1-10 of 1,000 rows Previous 1 2 3 4 5 ... 100 Next

- **Logs**: `log()`, `log2()`, `log10()`. Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude. They also convert multiplicative relationships to additive, a feature we'll come back to in modelling.

  All else being equal, I recommend using `log2()` because 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**: `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()`, which you'll learn about shortly.

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

Exercises

✓ Exercise 1

Currently `dep_time` and `sched_dep_time` are convenient to look at, but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight.

R Code | Start Over | Hint | Run Code | Submit Answer

```
transmute(flights,
  dep_time,
  dep_mins = ((dep_time %/% 100)*60) + (dep_time %% 100),
  sched_dep_time,
  sched_dep_mins = ((sched_dep_time %/% 100)*60) + (sched_dep_time %% 100)
)
```

dep_time <int>	dep_mins <dbl>	sched_dep_time <int>	sched_dep_mins <dbl>
517	317	515	315
533	333	529	329
542	342	540	340
544	344	545	345
554	354	600	360
554	354	558	358
555	355	600	360
557	357	600	360
557	357	600	360
558	358	600	360

1-10 of 1,000 rows   Previous 1 2 3 4 5 6 ... 100 Next

Good Job!

✓ Exercise 2

Compare `air_time` with `arr_time - dep_time`. What do you expect to see? What do you see? How do you explain this?

R Code Start Over Solution Run Code Submit Answer

```
flights <- mutate(flights, total_time = arr_time - dep_time)
flight_times <- select(flights, air_time, total_time)
filter(flight_times, air_time != total_time)
flight_times

# la resta de la hora de llegada menos la hora de salida no será igual a un valor en minutos
```

air_time <dbl>	total_time <int>
227	313
227	317
160	381
183	460
116	258
150	186
158	358
53	152
140	281
138	195

1-10 of 1,000 rows   Previous 1 2 3 4 5 6 ... 100 Next

air_time <dbl>	total_time <int>
227	313
227	317
160	381
183	460
116	258
150	186
158	358
53	152
140	281
138	195

1-10 of 1,000 rows   Previous 1 2 3 4 5 6 ... 100 Next

Good Job! it doesn't make sense to do math with `arr_time` and `dep_time` until you convert the values to minutes past midnight (as you did with `dep_time` and `sched_dep_time` in the previous exercise).

✓ Exercise 3

Compare `dep_time`, `sched_dep_time`, and `dep_delay`. How would you expect those three numbers to be related?

R Code | Start Over | Run Code

```r
select(flights, dep_time, sched_dep_time, dep_delay)

# dep_delay es una resta de dep_time menos sched_dep_time (La diferencia
# de la hora de salida agendada con la hora de salida real)
```

dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
517	515	2
533	529	4
542	540	2
544	545	-1
554	600	-6
554	558	-4
555	600	-5
557	600	-3
557	600	-3
558	600	-2

1-10 of 1,000 rows Previous **1** 2 3 4 5 6 ... 100 Next

Exercise 4

Find the 10 most delayed flights (`dep_delay`) using a ranking function. How do you want to handle ties? Carefully read the documentation for `min_rank()`.

R Code | Start Over | Hint | Run Code | Submit Answer

```r

filter(flights, rank(desc(dep_delay)) < 10)  %>%
arrange(desc(dep_delay))
```

year <int>	mo... <int>	d... <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>	arr_time <int>	sched_arr_time <int>	arr_delay <dbl>	carrier <chr>
2013	1	9	641	900	1301	1242	1530	1272	HA
2013	6	15	1432	1935	1137	1607	2120	1127	MQ
2013	1	10	1121	1635	1126	1239	1810	1109	MQ
2013	9	20	1139	1845	1014	1457	2210	1007	AA
2013	7	22	845	1600	1005	1044	1815	989	MQ
2013	4	10	1100	1900	960	1342	2211	931	DL
2013	3	17	2321	810	911	135	1020	915	DL

✓ Exercise 5

What does `1:3 + 1:10` return? Why?

R Code | Start Over | Hint | Run Code | Submit Answer

```
1:3 + 1:10

# Al terminar la longitud de 1:3, repite el conjunto de datos para los demás valores de 1:10
```

```
Warning in 1:3 + 1:10: longer object length is not a multiple of shorter object
length
```

```
 [1]  2  4  6  5  7  9  8 10 12 11
```

Nice! R repeats 1:3 three times to create a vector long enough to add to 1:10. Since the length of the new vector is not exactly the length of 1:10, R also returns a warning message.

✓ Exercise 6

What trigonometric functions does R provide? Hint: look up the help page for `Trig`.

R Code | Start Over | Run Code

```
?Trig
```

Previous Topic