Week 2 Demo

February 16, 2021

1 Let's recap:

R let's us assign single values to variables:

[1]:
```
a <- 445
b <- 67
c <- 2
```

It also let's us perform operations with variables or explicit values:

[2]:
```
5 + 89
445 + 67
a + b
a + c
a + b + c
b/c
a*c
```

94

512

512

447

514

33.5

890

It also let's us to combine multiple values into a single object which we call a **vector** with the **c()** function:

[3]:
```
vec <- c(5, 89, a, b, c)
```

Why did R let us combine raw values and variables? Because all 5 are numeric. Try executing this and think about how **vec0** is different from **vec**:

[4]:
```
vec0 <- c(5, 89, "a", "b", "c")
```

[5]:
```
vec0
```

1. '5' 2. '89' 3. 'a' 4. 'b' 5. 'c'

That's right - the quotes (' ') tell us that **vec0** values are treated as **strings**, or chr, whereas the elements in **vec** are **numbers**.

This is not all R is capable of. Sometimes we want to bundle together not just single values into a vector but several vectors into a...

2 Dataframe

Dataframes (df) are R's core data structure. Pretty much all datasets we work with from now on will be saved into R as dataframes. You can think of a dataframe **loosely** as an Excel spreadsheet, with **rows** and **columns**. Rows in a df are called **observations** and columns - **variables**. For starters, let's try replicating this mini dataset in R:

As the screenshot shows, there are two columns - **"name"** and **"age"**, and **4 rows**. We can think of this dataset as a collection of **2 vectors** - one for each column or variable. Then, we can start by declaring each vector, we've done similar things before:

[6]:
```
name <- c("Beth", "Raj", "Ken", "Jordyn")
age <- c(46, 51, 13, 2)
```

Let's check if both vectors look okay by **printing** them:

[7]:
```
name
age
```

1. 'Beth' 2. 'Raj' 3. 'Ken' 4. 'Jordyn'

1. 46 2. 51 3. 13 4. 2

So far so good. The only thing we need to figure out is how to bundle them together. Luckily, R has a function for that - **data.frame()** - we just need to think of a name for this new df and tell R which vectors it consists of:

[8]:
```
data <- data.frame(name, age)
```

We named the df **data** and assigned to it the 2 vectors, **name** and **age**. Let's check our work:

[9]:
```
data
```

A data.frame: 4 × 2

name	age
<chr>	<dbl>
Beth	46
Raj	51
Ken	13
Jordyn	2

Looks good! We are now ready to start analyzing our data. For starters, let's find out what the average age is. To do this, we will use the **mean()** function we are already familiar with. But we can't feed it the entire df, that's not going to work:

[10]:
```
mean(data)
```

```
Warning message in mean.default(data):
"argument is not numeric or logical: returning NA"
```

<NA>

It doesn't work because we are forcing R to calculate the average of a chr column - **name** which is coded as a bunch of text, and R doesn't quite know what to do with it. Instead, we need to zero in on the **age** column specifically. We do this in R by point the function to the specific column within the df we want to evaluate with the **$** sign:

[11]:
```
mean(data$age)
```

28

Now we're talking! Using the same logic, we can find the min, max, range, and length of this column, just like we did with individual vectors last week:

[12]:
```
min(data$age)
max(data$age)
range(data$age)
length(data$age)
```

2

51

1. 2 2. 51

4

We can even save some of these values for later use by assigning them to variables:

[13]:
```
minAge <- min(data$age)
maxAge <- max(data$age)
```

Try out some mathematical operations with our new vars:

[14]:
```
minAge + maxAge
```

53

[15]:
```
minAge * maxAge
```

102

Remember **subsetting**? We can try it on entire dfs as well, not just individual vectors. For example, the following code will only retain the records of people in our data younger than 20:

[16]:
```
youngDf <- data[data$age < 20,]

youngDf
```

A data.frame: 2 × 2

	name	age
	<chr>	<dbl>
3	Ken	13
4	Jordyn	2

Let's see how this works: we start out, as usual, by thinking of a name for our new object - **youngDf**. Then, we tell R we want to copy over the contents of the **data** df but only retain those records for which the value in the **age** column is less than 20. Even though it doesn't look like a big deal, the comma (**,**) after **20** is **really important**. It acts as a **separator** between the **rows** and **columns** of a df. To the **left** of the comma is the **row condition** and to the right - the **column one**. In this case then, we ask R to give us only the rows where age $<$ 20, and **ALL** the columns. Remember - if you leave the **column** condition after the comma empty, you essentially tell R you want all columns, which is exactly what we did.

Using the comma separator to distinguish between dataframe rows and columns enables us to access practically any "cell" of the dataframe = any intersection of a row and a column. For example, if we wanted to "reach" the name of Raj in the original df called **data**, we would need to select the second element (row=2) of the first column (column=1):

[17]:
```
data[2,1]
```

'Raj'

How about Raj's age? That would also be in the second row (row=2) but the second, rather than the first column (column=2):

[18]:
```
data[2,2]
```

51

Why does the following produce the same result:

[19]:
```
data$age[2]
```

51

What does the following line of code do:

[20]:
```
data[1:2,2]
```

1. 46 2. 51

The data we used so far can fit in a 3x2 dataframe. Real data however, may consist of a lot more observations (rows) and variables (columns). To get us started, we can take a look at datasets that come pre-installed with R - you can simply run **data()** in your console. For this exercise, we'll use a built-in R dataset called **mtcars**. We will first copy its contents to a dataframe called **myCars**:

[21]:
```
myCars <- mtcars
```

Let's explore the df:

[22]:
```
myCars
```

A data.frame: 32 × 11

	mpg <dbl>	cyl <dbl>	disp <dbl>	hp <dbl>	drat <dbl>	wt <dbl>	qsec <dbl>	vs <d
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1

We can see that there are 32 observations (rows) and 11 variables (columns). To further explore the variable composition, we can use two R functions: **summary()** and **str()**.

[23]:
```
summary(myCars)
```

```
      mpg             cyl             disp             hp       
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
 Median :19.20   Median :6.000   Median :196.3   Median :123.0  
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
      drat             wt             qsec             vs        
```

```
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

[24]:
```
str(myCars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 …
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 …
 $ disp: num  160 160 108 258 360 …
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 …
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 …
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 …
 $ qsec: num  16.5 17 18.6 19.4 17 …
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 …
 $ am  : num  1 1 1 0 0 0 0 0 0 0 …
 $ gear: num  4 4 4 3 3 3 3 4 4 4 …
 $ carb: num  4 4 1 1 2 1 4 2 2 4 …
```

While **str()** provides an overview of the data type of each variable (character, numeric, etc.), **summary()** gives us convenient descriptive statistics of numeric variables (mean, median, min, max, etc.).

Next, we can explore the meaning of each variable by consulting the R codebook associated with the dataset - it contains **metadata** (i.e. data about the data) about each variable:

[25]:
```
?mtcars
```

Let's take a look at the **mpg** column - we'd like to know the average mpg of cars in this dataset:

[26]:
```
mean(myCars$mpg)
```

20.090625

How about the highest/lowest mpg? We can save them into dedicated variables so we can use them later:

[27]:
```
maxMpg <- max(myCars$mpg)
minMpg <- min(myCars$mpg)
```

Right now, the dataset is not sorted on the **mpg column** - the mpg values are all over the place. We can use a combination of **subsetting** and the **order()** function to sort them:

[28]:
```
carsSorted <- myCars[order(myCars$mpg),]
```

[29]:
```
carsSorted
```

A data.frame: 32 × 11

	mpg <dbl>	cyl <dbl>	disp <dbl>	hp <dbl>	drat <dbl>	wt <dbl>	qsec <dbl>	vs <d
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1

Which car brand has the highest mpg? If we just use **max(myCars$mpg)**, we'd just get a number, with no additional information attached to it. Instead, we can use **which.max() to return the full row**:

[30]:
```
myCars[which.max(myCars$mpg),]
```

A data.frame: 1 × 11

	mpg <dbl>	cyl <dbl>	disp <dbl>	hp <dbl>	drat <dbl>	wt <dbl>	qsec <dbl>	vs <dbl>
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.9	1

Finally, let's review **conditional statements** - we can create an **if... else statement** which will check whether the **max mpg** in the dataset (remember we saved it earlier in a variable called **maxMpg**) is greater than 34 - this is the **logical test** part of the statement. If the answer is **TRUE**, i.e. yes, maxMpg is greater than 34, we'll ask R to print **"Yes!!!"**; if the answer is **FALSE**, i.e. maxMpg is NOT greater than 34, we'll print **"No... :("**:

[31]:
```
if (maxMpg > 34) "Yes!!!" else "No... :("
```

'No... :('