Dataframes

Vectors and matrices are super cool. However they don't address an important issue: holding multiple types of data and working with them at the same time. Dataframes are another special data structure that let's you handle large amounts and different types of data together. Because of this, they are generally the tool-of-choice for doing analyses in R.

We are going to focus on using dataframes using the `dplyr` package. `dplyr` comes as part of the `tidyverse` package bundle, you can install it with `install.packages("tidyverse")`. It can take awhile to install this on Linux, so perhaps start the command in another window while we go through the non-dplyr parts.

A small example

In a text editor, create the following example CSV file. We'll call it `cats.csv`.

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

Once we've saved it in the same directory we're working in, we can load it with `read.csv()`.

```
cats <- read.csv('cats.csv')
cats
```

```
##     coat weight likes_string
## 1 calico    2.1            1
## 2  black    5.0            0
## 3  tabby    3.2            1
```

Whenever we import a dataset with multiple types of values, R will autodetect this and make the output a dataframe. Let's verify this for ourselves:

```
class(cats)
```

```
## [1] "data.frame"
```

So, we've got a dataframe with multiple types of values. How do we work with it? Fortunately, everything we know about vectors also applies to dataframes.

Each column of a dataframe can be used as a vector. We use the `$` operator to specify which column we want.

```
cats$weight + 34
```

```
## [1] 36.1 39.0 37.2
```

```
class(cats$weight)
```

```
## [1] "numeric"
```

```
cats$coat
```

```
## [1] calico black  tabby
## Levels: black calico tabby
```

We can also reassign columns as if they were variables. The `cats$likes_string` likely represents a set of boolean value, lets update that column to reflect this fact.

```
class(cats$likes_string)  # before
```

```
## [1] "integer"
```

```
cats$likes_string <- as.logical(cats$likes_string)
class(cats$likes_string)
```

```
## [1] "logical"
```

We can even add a column if we want!

```
cats$age <- c(1, 6, 4, 2.5)
```

```
Error in `$<-.data.frame`(`*tmp*`, age, value = c(1, 6, 4, 2.5)) :
  replacement has 4 rows, data has 3
```

Notice how it won't let us do that. The reason is that dataframes must have the same number of elements in every column. If each column only has 3 rows, we can't add another column with 4 rows. Let's try that again with the proper number of elements.

```
cats$age <- c(1, 6, 4)
cats
```

```
##     coat weight likes_string age
## 1 calico    2.1         TRUE   1
## 2  black    5.0        FALSE   6
## 3  tabby    3.2         TRUE   4
```

Note that we don't have to call `class()` on every single column to figure out what they are. There are a number of useful summary functions to get information about our dataframe.

`str()` reports on the structure of your dataframe. It is an extremely useful function - use it on everything if you've loaded a dataset for the first time.

```
str(cats)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ coat        : Factor w/ 3 levels "black","calico",..: 2 1 3
##  $ weight      : num  2.1 5 3.2
##  $ likes_string: logi  TRUE FALSE TRUE
##  $ age         : num  1 6 4
```

As with matrices, we can use `dim()` to know how many rows and columns we're working with.

```
dim(cats)
```

```
## [1] 3 4
```

```
nrow(cats)  # number of rows only
```

```
## [1] 3
```

```
ncol(cats)  # number of columns only
```

```
## [1] 4
```

Factors

When we ran `str(cats)`, you might have noticed something weird. `cats$coat` is listed as a "factor". A factor is a special type of data that's *almost* a string.

It prints like a string (sort of):

```
cats$coat
```

```
## [1] calico black  tabby
## Levels: black calico tabby
```

It can be used like a string:

```
paste("The cat is", cats$coat)
```

```
## [1] "The cat is calico" "The cat is black"  "The cat is tabby"
```

But it’s not a string! The output of `str(cats)` gives us a clue to what’s actually happening behind-the-scenes.

```
str(cats)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ coat        : Factor w/ 3 levels "black","calico",..: 2 1 3
##  $ weight      : num  2.1 5 3.2
##  $ likes_string: logi  TRUE FALSE TRUE
##  $ age         : num  1 6 4
```

`str()` reports that the first values are 2, 1, 3 (and not text). Let’s use `as.numeric()` to reveal its true form!

```
as.numeric(cats$coat)
```

```
## [1] 2 1 3
```

```
cats$coat
```

```
## [1] calico black  tabby
## Levels: black calico tabby
```

A factor has two components, its levels and its values. Levels represent all possible values for a column. In this case, there’s only 3 possiblities: `black`, `calico` and `tabby`.

The actual values are 2, 1, and 3. Each value matches up to a specific level. So in our example, the first value is 2, which corresponds to the second level, `calico`. The second value is 1, which matches up with the first level, `black`.

Factors in R are a method of storing text information as one of several possible “levels”. R converts text to factors automatically when we import data, like from a CSV file. We’ve got several options here:

Convert the factor to a character vector ourselves:

```
cats$coat <- as.character(cats$coat)
class(cats$coat)
```

```
## [1] "character"
```

Tell R to simply not convert things to factors when we import it (`as.is=TRUE` is the R equivalent of “don’t touch my stuff!”):

```
new_cats <- read.csv('cats.csv', as.is=TRUE)
class(new_cats$coat)
```

```
## [1] "character"
```

Use the `read_csv()` function from the `readr` package. `readr` is part of the `tidyverse` and has a number of ways of reading/writing data with more sensible defaults.

```
library(tidyverse)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Conflicts with tidy packages ----------------------------------------------
```

```
## filter(): dplyr, stats
## lag():    dplyr, stats
```

```
even_newer_cats <- read_csv('cats.csv')
```

```
## Parsed with column specification:
## cols(
##   coat = col_character(),
##   weight = col_double(),
##   likes_string = col_integer()
## )
```

```
class(even_newer_cats$coat)
```

```
## [1] "character"
```

Performance considerations

As you can see, factors can be kind of a pain to deal with. So why do they even exist? The short answer is that they are an effective way of optimizing memory usage.

To demonstrate this, we'll examine the gapminder example dataset (`install.packages("gapminder")`).

```
library(gapminder)
head(gapminder)
```

```
## # A tibble: 6 x 6
##       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
## 4 Afghanistan      Asia  1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia  1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia  1977  38.438 14880372  786.1134
```

```
str(gapminder)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 22227415
##  $ gdpPercap: num  779 821 853 836 740 ...
```

Notice how `gapminder` contains several columns with a lot of highly repetitive data. For instance, the `continent` column contains only 5 values:

```
unique(gapminder$continent)
```

```
## [1] Asia     Europe   Africa   Americas Oceania
## Levels: Africa Americas Asia Europe Oceania
```

In this case, `gapminder$continent` has been stored as a factor. Let's examine the amount of space used if this column was stored as a character vector vs. storing the data as a factor.

```
library(pryr)
```

```
##
## Attaching package: 'pryr'
```

```
## The following objects are masked from 'package:purrr':
##
##     compose, partial
```

```
object_size(gapminder$continent)
```

```
## 7.51 kB
```

```
object_size(as.character(gapminder$continent))
```

```
## 13.9 kB
```

The character version of `gapminder$continent` takes up almost twice as much space! Storing things as one of several possible integer values behind the scenes is a lot more efficient than storing the entire set of text for every single entry. Note that the amount of memory saved depends on the repetitiveness of the data. If a column has a lot of unique text values, converting it to a factor will likely be of little benefit.

The takeaway here is that if you ever find yourself working on a large dataset and memory usage becomes an issue, converting your most repetitive string columns to factors can be a useful way to save space. Otherwise, just use character vectors (less hidden "gotchas" that way).

Next section