Computing by groups within data.frames with plyr

- Install and load `plyr`
- Load the Gapminder data
- Introduction to `ddply()`
- Recall the function we wrote to fit a linear model
- Make the function available in the workspace
- Apply our function inside ddply
- References

This material was last used in 2014. Since then, we are turning more to `dplyr`, including for data aggregation. But `plyr` is still a very useful package and so we leave this material here.

Install and load `plyr`

If you have not already done so, you'll need to install `plyr`.

```
install.packages("plyr", dependencies = TRUE)
```

We also need to load it.

```
library(plyr)
```

Note: if you are using both `plyr` and `dplyr` in a script, you should always load `plyr` first, then `dplyr`.

`plyr` Big Ideas

The `plyr` functions will not make much sense viewed individually, e.g. simply reading the help for `ddply()` is not the fast track to competence. There is a very important over-arching logic for the package and it is well worth reading the article The split-apply-combine strategy for data analysis (http://www.jstatsoft.org/v40/i01/paper). Though it is no substitute for reading the above, here is the most critical information:

- **split-apply-combine**: A common analytical pattern is to split data into pieces, apply some function to each pieces, and combine the results back together again. Recognize when you're solving such a problem and exploit the right tools.
- The computations on these pieces must be truly independent, i.e. the problem must be embarrassingly or pleasingly parallel (http://en.wikipedia.org/wiki/Embarrassingly_parallel), in order to use `plyr`.
- The heart of `plyr` is a set a functions with names like this: `XYply` where `X` specifies what sort of input you're giving and `Y` specifies the sort of output you want.
    - `a` = array, where matrices and vectors are important special cases
    - `d` = data.frame
    - `l` = list
    - `_` = no output; only valid for `Y`, obviously; useful when you're operating on a list purely for the side effects, e.g., making a plot or sending output to screen/file
- The usage is very similar across these functions. Here are the main arguments:
    - `.data` is the first argument = the input
    - the next argument specifies how to split up the input into pieces; it does not exist when the input is a list, because the pieces must be the list components
    - then comes the function and further arguments needed to describe the computation to be applied to the pieces

Today we emphasize `ddply()` which accepts a data.frame, splits it into pieces based on one or more factors, computes on the pieces, then returns the results as a data.frame. For the record, the base R functions most relevant to `ddply()` are `tapply()` and friends.

Load the Gapminder data

As usual, load the Gapminder excerpt.

```
library(gapminder)
```

Introduction to `ddply()`

Let's say we want the maximum life expectancy for each continent. We're providing a data.frame as input and we want a data.frame as output. Therefore this is a job for `ddply()` . We want to divide the data.frame into pieces based on `continent` . Here's the call

```
(max_le_by_cont <-
   ddply(gapminder, ~ continent, summarize, max_le = max(lifeExp)))
##   continent max_le
## 1    Africa 76.442
## 2  Americas 80.653
## 3      Asia 82.603
## 4    Europe 81.757
## 5   Oceania 81.235
```

Let's study the return value.

```
str(max_le_by_cont)
## 'data.frame':    5 obs. of  2 variables:
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 1 2 3 4
5
##  $ max_le   : num  76.4 80.7 82.6 81.8 81.2
levels(max_le_by_cont$continent)
## [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
```

We got a data.frame back, with one observation per continent, and two variables: the maximum life expectancies and the continent, as a factor, with the same levels in the same order, as for the input data.frame `gapminder`. If you have sweated to do such things with base R, this minor miracle might make you cry tears of joy (or anguish over all the hours you have wasted.)

`summarize()` or its synonym `summarise()` is a function provided by `plyr` that creates a new data.frame from an old one. It is related to the built-in function `transform()` that transforms variables in a data.frame or adds new ones. Feel free to play with it a bit in some top-level commands; you will use it alot inside `plyr` calls.

The two variables in `max_le_by_cont` come from two sources. The `continent` factor is provided by `ddply()` and represents the labelling of the life expectancies with their associated continent. This is the book-keeping associated with dividing the input into little bits, computing on them, and gluing the results together again in an orderly, labelled fashion. We can take more credit for the other variable `max_le`, which has a name we chose and arises from applying a function we specified (`max()`) to a variable of our choice (`lifeExp`).

You try: compute the minimum GDP per capita by continent. Here's what I get:

```
##   continent  min_gdppc
## 1    Africa   241.1659
## 2  Americas  1201.6372
## 3      Asia   331.0000
## 4    Europe   973.5332
## 5   Oceania 10039.5956
```

You might have chosen a different name for the minimum GDP/capita's, but your numerical results should match.

The function you want to apply to the continent-specific data.frames can be built-in, like `max()` above, or a custom function you've written. This custom function can be written in advance or specified 'on the fly'. Here's one way to count the number of countries in this dataset for each continent.

```
ddply(gapminder, ~ continent,
      summarize, n_uniq_countries = length(unique(country)))
##   continent n_uniq_countries
## 1    Africa               52
## 2  Americas               25
## 3      Asia               33
## 4    Europe               30
## 5   Oceania                2
```

Here is another way to do the same thing that doesn't use `summarize()` at all:

```
ddply(gapminder, ~ continent,
      function(x) c(n_uniq_countries = length(unique(x$country))))
##   continent n_uniq_countries
## 1    Africa               52
## 2  Americas               25
## 3      Asia               33
## 4    Europe               30
## 5   Oceania                2
```

In pseudo pseudo-code, here's what's happening above:

```
returnValue <- an empty receptacle with one "slot" per country
for each possible country i {
    x  <- subset(gapminder, subset = country == i)
    returnValue[i] <- length(unique(x$country))
    name or label for returnValue[i] is set to country i
}
ddply packages returnValue and associated names/labels as a nice dat
a.frame
```

You don't have to compute just one thing for each sub-data.frame, nor are you limited to computing on just one variable. Check this out.

```
ddply(gapminder, ~ continent, summarize,
      min_le = min(lifeExp), max_le = max(lifeExp),
      med_gdppc = median(gdpPercap))
##   continent min_le max_le med_gdppc
## 1    Africa 23.599 76.442  1192.138
## 2  Americas 37.579 80.653  5465.510
## 3      Asia 28.801 82.603  2646.787
## 4    Europe 43.585 81.757 12081.749
## 5   Oceania 69.120 81.235 17983.304
```

Recall the function we wrote to fit a linear model

We recently learned how to write our own R functions (Part 1 (block011_write-your-own-function-01.html), Part 2 (block011_write-your-own-function-02.html), Part 3 (block011_write-your-own-function-03.html)). We then wrote a function (block012_function-regress-lifeexp-on-year.html) to use on the Gapminder dataset. This function `le_lin_fit()` takes a data.frame and expects to find variables for life expectancy and year. It returns the estimated coefficients from a simple linear regression. We wrote it with the goal of applying it to the data from each country in Gapminder. That's what we do here.

Make the function available in the workspace

Define the function developed elsewhere (block012_function-regress-lifeexp-on-year.html):

```
le_lin_fit <- function(dat, offset = 1952) {
  the_fit <- lm(lifeExp ~ I(year - offset), dat)
  setNames(coef(the_fit), c("intercept", "slope"))
}
```

Let's try it out on the data for one country, just to reacquaint ourselves with it.

```
le_lin_fit(subset(gapminder, country == "Canada"))
##  intercept      slope
## 68.8838462  0.2188692
```

Apply our function inside ddply

We are ready to scale up to **all countries** by placing this function inside a `ddply()` call.

```
j_coefs <- ddply(gapminder, ~ country, le_lin_fit)
str(j_coefs)
## 'data.frame':    142 obs. of  3 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 2 3 4 5 6
7 8 9 10 ...
##  $ intercept: num  29.9 59.2 43.4 32.1 62.7 ...
##  $ slope    : num  0.275 0.335 0.569 0.209 0.232 ...
tail(j_coefs)
##                country intercept       slope
## 137          Venezuela  57.51332  0.32972168
## 138            Vietnam  39.01008  0.67161538
## 139 West Bank and Gaza  43.79840  0.60110070
## 140        Yemen, Rep.  30.13028  0.60545944
## 141             Zambia  47.65803 -0.06042517
## 142           Zimbabwe  55.22124 -0.09302098
```

We did it! By the time we've packaged the computation in a function, the call itself is deceptively simple. To review, here's the script I would save from our work in this tutorial:

```
library(plyr)
library(gapminder)
le_lin_fit <- function(dat, offset = 1952) {
  the_fit <- lm(lifeExp ~ I(year - offset), dat)
  setNames(coef(the_fit), c("intercept", "slope"))
}
j_coefs <- ddply(gapminder, ~ country, le_lin_fit)
```

That's all. After we've developed the `le_lin_fit()` function and gotten to know `ddply()`, this task requires about 5 lines of R code.

Reflect on how incredibly convenient this is. If you're coming from another analytical environment, how easy would this task have been? If you had been asked to do this with R a week ago, would you have written a `for` loop or given up? The take away message is this: if you are able to write custom functions, the `plyr` package can make you extremely effective at computing on pieces of a data structure and reassembling the results.

References

`plyr` paper: The split-apply-combine strategy for data analysis (http://www.jstatsoft.org/v40/i01/paper), Hadley Wickham, Journal of Statistical Software, vol. 40, no. 1, pp. 1–29, 2011. Go here (http://www.jstatsoft.org/v40/i01/) for supplements, such as example code from the paper.