Class Code

July 29, 2015

Some data structures and recursion! First, some alternate ways to represent data:

```
rm(list = ls())

chars <- list("A", "B", "C", c(1,2,3), list("X", "Y"), NULL)
print(chars)
print(chars[[6]])
print(is.null(chars[[6]]))
print(is.null(chars[[5]]))
print(is.na(chars[[5]]))

char <- list("A", NULL)  # data, "rest"

chars <- list("A", "B")
chars <- list("A", list("B", NULL))  # data, "rest"
chars <- list("A", list("B", list("C", NULL)))
```

And then the "simple" and "elegant" ways to print data from the "data, rest" model:

```
data <- chars
while(!is.null(data)) {
  el <- data[[1]]
  print(el)
  data <- data[[2]]
}


print_list <- function(data) {
  el <- data[[1]]
  rest <- data[[2]]

  if(is.null(rest)) {
    print(el)
  } else {
    print(el)        # what if the order of these two
    print_list(rest) # lines were switched?
  }
}

print_list(chars)
```

A nicer way to extract an element by index:

```
# todo: fix this so it doesn't error if n is too large ;)
get_el <- function(data, n) {
  el <- data[[1]]
  rest <- data[[2]]

  if(n == 1) {
    return(el)
  } else {
    new_index <- n - 1
    answer <- get_el(rest, new_index)
    return(answer)
  }
}

print(get_el(chars, 2))
print(get_el(chars, 3))
```

Appending a new element:

```
append_end <- function(data, new_el) {
  el <- data[[1]]
  rest <- data[[2]]

  if(is.null(rest)) {
    new_rest <- list(new_el, NULL)
    return(list(el, new_rest))
  } else {
    new_rest <- append_end(rest, new_el)
    return(list(el, new_rest))
  }
}

chars2 <- append_end(chars, "D")
print_list(chars2)
chars3 <- append_end(chars2, "B")
print_list(chars3)
```

Our imagination is the limit in terms of what sorts of operations we can do with these structures. Here's our own version of `lapply()`:

```
mylapply <- function(data, func, ...) {
  el <- data[[1]]
  rest <- data[[2]]

  if(is.null(rest)) {
    answer <- list(func(el, ...), NULL)
    return(answer)
  } else {
    answer_rest <- mylapply(rest, func, ...)
    answer <- list(func(el, ...), answer_rest)
    return(answer)
  }
}

print(tolower("C"))    # tolower() converts strings to lower case
adjusted <- mylapply(chars, tolower)
print_list(adjusted)
```

The above describes *linked lists*.

What about lists with "left, data, right" model? A *tree*. (See figure in "Something fun for July 1" on the documents page.) With the rule that smaller elements go left, a *binary search tree*.

```
t <- list(list(NULL, "A", NULL), "C", list(list(NULL, "D", NULL), "F", NULL))


print_tree <- function(data) {
  left <- data[[1]]
  el <- data[[2]]
  right <- data[[3]]

  if(!is.null(left)) {
    print_tree(left)
  }
  print(el)
  if(!is.null(right)) {
    print_tree(right)
  }
}


print_tree(t)
```

Inserting, breaking down into cases with two sub-cases each.

```
insert_tree <- function(data, new_el) {
  left <- data[[1]]
  el <- data[[2]]
  right <- data[[3]]

  if(new_el < el) {   # goes left
    if(is.null(left)) {  # insert there
      new_left <- list(NULL, new_el, NULL)
      answer <- list(new_left, el, right)
      return(answer)
    } else {
      new_left <- insert_tree(left, new_el)
      answer <- list(new_left, el, right)
      return(answer)
    }
  } else {            # goes right
    if(is.null(right)) {  # insert there
      new_right <- list(NULL, new_el, NULL)
      answer <- list(left, el, new_right)
      return(answer)
    } else {
      new_right <- insert_tree(right, new_el)
      answer <- list(left, el, new_right)
      return(answer)
    }
  }
}

t2 <- insert_tree(t, "B")
print_tree(t2)
t3 <- insert_tree(t2, "E")
print_tree(t3)
```

July 27, 2015

We finished up our discussion of PCA including some tricks for running and plotting PCA in R (see files for the 24th), and we discussed very basic project management with Git.

July 24, 2015

We looked at some PCA. You can find the R-markdown here and the html file here.

July 22, 2015

Application time! We briefly discussed the functions `prod()`, `sum()`, `dnorm()`, `pnorm()` and `qnorm()` (as well as "special" numeric values `NaN` and `Inf`) before defining the "likelihood" of a simple being from a given distribution. (Of course `prod()` and `sum()` we could write ourselves, if we didn't know that R already had them.)

```
rm(list = ls())


sample <- c(16, 20, 24)
lhoodsa <- dnorm(sample, mean = 20, sd = 4)
lhoodsb <- dnorm(sample, mean = 100, sd = 10)
lhooda <- prod(lhoodsa)
lhoodb <- prod(lhoodsb)
print(lhooda)
print(lhoodb)
```

Likelihood is cool, but we run into precision issues with the very small numbers. Enter the negative log-likelihood (a thing that is smaller when likelihood is larger):

```
nll_sizes <- function(params, data) {
  propa <- params[1]
  propb <- 1 - propa
  mua <- params[2]
  sigmaa <- params[3]
  mub <- params[4]
  sigmab <- params[5]

  if(propa < 0 | propa > 1) {
    return(Inf)
  }

  lhoods <- propa * dnorm(data, mean = mua, sd = sigmaa) +
            propb * dnorm(data, mean = mub, sd = sigmab)

  nll <- -1 * sum(log(lhoods))
  return(nll)
}
```

We even wrote a function to compute NLL of a sample given parameters in a vector and the sample:

```
nll_norm <- function(model_params, data) {
  mu <- model_params[1]
  sigma <- model_params[2]

  lhoods <- dnorm(data, mean = mu, sd = sigma)
  nll <- -1 * sum(log(lhoods))
  return(nll)
}
```

And then we used `optim()` to find the minimum NLL of the function by tweaking a parameters vector:

```
sample <- rnorm(1000, mean = 20, sd = 4)
startparams <- c(1,1) # guess for mu, sigma
ml <- optim(startparams, nll_norm, data = sample)
print(ml)
```

In this situation the "maximum-likelihood-estimators" are available in closed form as `mean(sample)` and `sd(sample)`, but this isn't always the case...

We looked at an example of a mixed sample of butterfly sizes from two subspecies:

```
library(ggplot2)
sizes <- read.table("../../data/sizes.txt",
                    header = TRUE,
                    stringsAsFactors = FALSE,
                    sep = "\t")
p <- ggplot() +
  stat_bin(data = sizes, mapping = aes(x = size))
plot(p)
```

(The data file can be found **here**.)

What we don't know is what proportion of the sample are subspecies A, what the mean and sd of subsp A is, or the mean or sd of subsp B. So we wrote a function that computes NLL given estimates of these parameters and a sample:

```
nll_sizes <- function(params, data) {
  propa <- params[1]
  propb <- 1 - propa
  mua <- params[2]
  sigmaa <- params[3]
  mub <- params[4]
  sigmab <- params[5]

  if(propa < 0 | propa > 1) {
    return(Inf)
  }

  lhoods <- propa * dnorm(data, mean = mua, sd = sigmaa) +
            propb * dnorm(data, mean = mub, sd = sigmab)

  nll <- -1 * sum(log(lhoods))
  return(nll)
}
```

Using the maximum-likelihood approach to estimate the parameters:

```
startparams <- c(0.5, 50, 40, 130, 30)
# startparams <- c(0.5, 100, 30, 100, 30)
ml <- optim(startparams, nll_sizes, data = sizes$size)
str(ml)
```

Note that in this case the `optim()` search is pretty sensitive to the initial parameter vector; uncommenting the commented line above gave bad results.

We didn't have time to talk about confidence intervals using approaches like this, but for those interested check out Ecological Models and Data in R, especially Chapter 6.

July 20, 2015

Today was all about exploring the S3-object system in R. Major terms are *generic functions*, which *dispatch* to *methods* based on the `"class"` attribute of the first parameter (aka the object). A *constructor* returns a newly built object with its `"class"` attribute set. (Conceptually, objects are data + methods associated with it via class, and class is the "blueprint" for objects: essentially the constructor + method functions themselves.)

We started by examining the effects of the `"class"` attribute for some lists returned by R functions (`lm()` and `anova()`). First, R's default behavior:

```
treatments <- c("w", "w", "c", "c")
heights <- c(4.2, 5.4, 2.1, 3.2)
lm_res <- lm(heights ~ treatments)
anova_res <- anova(lm_res)
print(lm_res)
print(anova_res)

str(lm_res)
str(anova_res)
```

Next, mucking about with the `'"class"` attribute:

```
attr(lm_res, "class") <- NULL
# preferred equivalent:
lm_res <- unclass(lm_res)
print(lm_res)

# what happens if we set the class attribute incorrectly?
class(anova_res) <- "lm"
print(anova_res)
class(anova_res) <- c("anova", "data.frame")
print(anova_res)
```

The `print()` function is generic, and so is `plot()`; they both dispatch to different methods depending on the class of the object (e.g., `plot()` to `plot.lm()` for data with `"class"` of `"lm"`).

```
a <- rnorm(100, mean= 10, sd = 2)
b <- a + rnorm(100, mean = 1, sd = 0.5)
plot(a, b)
lm_res <- lm(b ~ a)
plot(lm_res)
```

To explore this idea, we created our own class type, called `"truncated_normal_sample"` using the code from yesterday:

```
trunc_rnorm <- function(lower, upper, count, mean, sd) {
  sample <- rnorm(count, mean = mean, sd = sd)
  loop_counter <- 0
  while(any(sample < lower | sample > upper)) {
    badl <- sample < lower | sample > upper
    bad_vals <- sample[badl]
    bad_count <- length(bad_vals)
    new_vals <- rnorm(bad_count, mean = mean, sd = sd)
    sample[badl] <- new_vals

    loop_counter <- loop_counter + 1
    if(loop_counter > 100000) {
      return(NA)
    }
  }
  return(sample)
}


truncated_normal_sample <- function(lower, upper, count, mean, sd) {
  obj <- list()
  obj$sample <- trunc_rnorm(lower, upper, count, mean, sd)
  obj$lower <- lower
  obj$upper <- upper
  obj$original_mean <- mean
  obj$original_sd <- sd
  class(obj) <- "truncated_normal_sample"
  return(obj)
}


trsamp <- truncated_normal_sample(0, 30, 25, 20, 10)

print(class(trsamp))
print(trsamp)
```

The default `print.default()` is ugly, so we defined our own method that `print()` can dispatch to:

```
print.truncated_normal_sample <- function(obj) {
  print("Truncated normal sample, limited to:")
  print(c(obj$lower, obj$upper))
  print("Original sampling mean and sd:")
  print(c(obj$original_mean, obj$original_sd))
  print("First 10 elements:")
  print(head(obj$sample, n = 10))
}

print(trsamp)
```

That's better. Similarly, we want to define a `mean.truncated_normal_sample()` so we can get the mean of the object's sample:

```
print(mean(trsamp)) # no worky

mean.truncated_normal_sample <- function(obj) {
  return(mean(obj$sample))
}

print(mean(trsamp)) # worky
```

And maybe we want a `limitrange()` function, which returns difference in the upper and lower values for the object. In this case, the generic doesn't already exist, so we have to define the method *and* the generic.

```
print(limitrange(trsamp)) # no worky

# method:
limitrange.truncated_normal_sample <- function(obj) {
  return(obj$upper - obj$lower)
}

print(limitrange(trsamp)) # still no worky

# generic:
limitrange <- function(obj) {
  UseMethod("limitrange", obj) # dispatch happens here based on class attr
}

print(limitrange(trsamp)) # worky
```

July 17, 2015

While loops!

```
counter <- 1
while(counter < 4) {
  print("counter is")
  print(counter)
}
print("done")
```

Don't forget about `any()` and `all()`, since only the first element of a logical vector is used for the check in a while or if:

```
counter <- c(1,100)
while(any(counter < 4)) {
  print("counter is:")
  print(counter)
}
print("done"
```

We wrote a function that uses while-loops and if-statements to generate a truncated normal sample, returning `NA` if more than 100K loopings are necessary. Here's the final version and its usage:

```
trunc_norm <- function(lower, upper, count, mean, sd) {
  sample <- rnorm(count, mean = mean, sd = sd)
  loop_counter <- 0
  while(any(sample < lower | sample > upper)) {
    badl <- sample < lower | sample > upper
    bad_vals <- sample[badl]
    bad_count <- length(bad_vals)
    new_vals <- rnorm(bad_count, mean = mean, sd = sd)
    sample[badl] <- new_vals

    loop_counter <- loop_counter + 1
    if(loop_counter > 100000) {
      return(NA)
    }
  }
  return(sample)
}



sample_trunc <- trunc_norm(15, 15.0001, 1000, 20, 5)
if(!is.na(sample_trunc)) {
  hist(sample_trunc)
} else {
  print("oops, took too long")
}
```

For loops:

```
ids <- c("AGP", "CYP6B", "ALQR")
for(id_el in ids) {
  print("id_el is now:")
  print(id_el)
}
print("done")
```

Are for loops intrinsically slow? A couple tests are revealing:

```
counter <- 0
for(i in seq(1,1000000)) {
  counter <- counter + 1
  if(counter %% 1000 == 0) {
    print(counter)
  }
}


counter <- 0
for(i in seq(1,1000000)) {
  counter <- c(counter, 1)
  if(length(counter%%1000 == 0)) {
    print(length(counter))
  }
}
```

Returning to functions, we wrote a "functional" version of our random-sample-truncator:

```
sample_trunc <- function(lower, upper, count, sample_func, ...) {
  sample <- sample_func(count, ...)
  loop_counter <- 0
  while(any(sample < lower | sample > upper)) {
    badl <- sample < lower | sample > upper
    bad_vals <- sample[badl]
    bad_count <- length(bad_vals)
    new_vals <- sample_func(bad_count, ...)
    sample[badl] <- new_vals

    loop_counter <- loop_counter + 1
    if(loop_counter > 100000) {
      return(NA)
    }
  }
  return(sample)
}

s <- sample_trunc(15, 30, 1000, rnorm, mean = 20, sd = 5)
hist(s)
s <- sample_trunc(1, 4, 1000, rexp, rate = 0.5)
hist(s)
```

July 15, 2015

First we talked about a quick gotcha: `a_vector[sel]` returns a vector (possible of length 1). Similarly, `a_list[sel]` returns a list (possibly with only one element), hence the existence of `[[]]` to "pull out" an element from a list. Data frames are a kind of list, so `df[sel]` returns a data frame (possibly with only one column). Breaking convention though, `df[row_sel, col_sel]` returns a data frame, *unless* that data frame would have only one column, in which case a vector is returned instead.

We also looked at `paste()` and `str_c()`:

```
firsts <- c("Shawn", "James")
lasts <- c("O'Neil", "Regier")

fulls <- paste(firsts, lasts, sep = " ")
print(fulls)

library(stringr)
fulls <- str_c(firsts, lasts, sep = "")
print(fulls)
```

And we looked at `summarize()` from the `dplyr` package, and simpler alternative to `do()` for some cases:

```
library(dplyr)

fish <- read.table("../../data/fish.txt",
                   header = TRUE,
                   sep = "\t",
                   stringsAsFactors = FALSE)

print(fish)

mean_cov_weight <- function(sub_df, remove_nas = FALSE) {
  weights <- sub_df$weight
  meanw = mean(weights, na.rm = remove_nas)
  covw = sd(weights, na.rm = remove_nas)/meanw
  ret_df <- data.frame(mean_weight = meanw, cov_weight = covw)
  return(ret_df)
}

fish_by_species <- group_by(fish, species)
stats_by_species <- do(fish_by_species, mean_cov_weight(.))
print(stats_by_species)


stats_by_species <- summarize(fish_by_species,
                              mean_weight = mean(weight),
print(stats_by_species)
```

We talked about `gather()` from the `tidyr` package:

```
geno_wide <- read.table("../../data/small_wide.txt",
                        header = TRUE,
                        sep = "\t",
                        stringsAsFactors = FALSE)


print(head(geno_wide))


library(tidyr)
geno_long <- gather(geno_wide, genotype, expression, C6, L4)
print(head(geno_long, n = 20))
```

(The `small_wide.txt` file can be found **here**.)

For the larger dataset with lots of columns that need gathering (**good_expr_wide.txt**), we used some syntactic sugar to specify the columns that *don't* need gathering:

```
expr_sample_only <- gather(expr_wide, sample, expression, -id, -annotation)
print(head(expr_sample_only))
```

And the `extract()` function from `tidyr`, which uses the language regular expressions to split a column up into other columns:

```
expr <- extract(expr_sample_only,
                sample,
                c("genotype", "treatment", "tissue", "rep"),
                regex = "(C6|L4)_(chemical|control)_(A|B|C)(1|2|3)")

print(head(expr))
```

And joining data frames on common columns with `merge()`:

```
heights <- data.frame(first = c("Joe", "Mary", "Brent", "Karen"),
                      last = c("Withmore", "O'Leary", "Liston", "Streeter"),
                      height = c(5.5, 5.2, 6.1, 5.3))

ages <- data.frame(first = c("Brent", "Joe", "Karen", "Chris"),
                   last = c("Liston", "Jones", "Streeter", "Peterson"),
                   age = c(27, 19, 22, 34))

print(heights)
print(ages)

a <- merge(heights, ages)
a <- merge(heights, ages, by = c("first"))
a <- merge(heights, ages, all = TRUE)
```

July 13, 2015

We looked at passing optional parameters to a function using `...` in `lapply()`. Also the `unlist()` function.

```
# our own function
ipercentile_range <- function(x, lower = 0.25, upper = 0.75) {
  lower_val <- quantile(x, lower)
  upper_val <- quantile(x, upper)
  return(upper_val - lower_val)
}

irq1 <- ipercentile_range(sample$s1)  # default
irq1_90 <- ipercentile_range(sample$s1, lower = 0.05, upper = 0.95)
print(irq1)
print(irq1_90)

inter_90s <- lapply(sample, ipercentile_range, lower = 0.05, upper = 0.95)
inter_90s <- unlist(inter_90s)
print(inter_90s)
```

Syntactic sugar, infix vs. prefix, redefining and defining infix functions:

```
# functions are everywhere
val <- `+`(c(1,4), c(3,7))
print(val)

# probably not a good idea:
`+` <- function(a, b) {
  val <- a ^ b
  return(val)
}

print(`+`(3,7))
print(3 + 7)

# a better idea:
`%equalish%` <- function(a, b) {
  res <- abs(a - b) < 0.0000001
  }

x <- c(4, 2)
y <- c(4.00001, 6)
print(`%equalish`(x, y))
print(x %equalish% y)
```

The `dplyr` library, `group_by()`: (The fish table can be found **here**.)

```
library(dplyr)

fish <- read.table("../../data/fish.txt",
                   header = TRUE,
                   sep = "\t",
                   stringsAsFactors = FALSE)

print(fish)

fish_by_species <- group_by(fish, species)
print(fish_by_species)
print(class(fish_by_species))
print(attr(fish_by_species, "vars"))
```

A function that takes a data frame, and returns columns for the mean weight and the coefficient of variation of the weight (sd/mean; in class I **incorectly** defined it as mean/sd). It also takes an optional `remove_nas` parameter that can be passed onto `mean()` and `sd()` for their `na.rm` parameters.

```
mean_cov_weight <- function(sub_df, remove_nas = FALSE) {
  weights <- sub_df$weight
  meanw = mean(weights, na.rm = remove_nas)
  covw = sd(weights, na.rm = remove_nas)/meanw
  ret_df <- data.frame(mean_weight = meanw, cov_weight = covw)
  return(ret_df)
}
```

And `do()`. Note that `do()` requires we specify the "form" of the functional call, using `.` in place of the sub-dataframe group parameter. A type of syntactic sugar.

```
weight_stats_by_species <- do(fish_by_species, mean_cov_weight(.))
print(weight_stats_by_species)

fish_by_sp_lake <- group_by(fish, species, lake)
stats_by_sp_lake <- do(fish_by_sp_lake, mean_cov_weight(.))
print(stats_by_sp_lake)
```

Using `do()` with optional parameters:

```
weight_stats_by_species <- do(fish_by_species,
                              mean_cov_weight(., remove_nas = TRUE))
print(weight_stats_by_species)
```

And having a function return a data frame with multiple rows, for e.g. computing a per-group mean-normalized column:

```
mean_norm_weight <- function(subdf) {
  ret_df <- subdf
  ret_df$norm_weight <- ret_df$weight - mean(ret_df$weight)
  return(ret_df)
}

norm_group <- do(fish_by_species, mean_norm_weight(.))
print(norm_group)
```

And we used these concepts to move further in our class example.

```
library(dplyr)

expr <- read.table("../../data/expr_long_2_fixed.txt",
                   header = TRUE,
                   stringsAsFactors = FALSE,
                   sep = "\t",
                   comment.char = "")

# subsample just 100 genes for speed of demonstration
first_100 <- unique(expr$id)[seq(1,100)]
expr100 <- expr[expr$id %in% first_100, ]

# modified function for p-value computation, taking the formula as an
# optional parameter
subdf_to_pvals_df <- function(subdf, form = expression ~ genotype +
                                                         treatment +
                                                         genotype:treatment) {
  lm1 <- lm(form,
            data = subdf)
  anova1 <- anova(lm1)

  pvals1 <- anova1$"Pr(>F)"
  pvals1 <- as.list(pvals1)
  pvals1_df <- data.frame(pvals1)
  colnames(pvals1_df) <- rownames(anova1)
  return(pvals1_df)
}

# by id
expr_by_id <- group_by(expr100, id)
results <- do(expr_by_id, subdf_to_pvals_df(.))
print(head(results))

# by annotation
expr_by_annotation <- group_by(expr100, annotation)
results <- do(expr_by_annotation, subdf_to_pvals_df(.))
print(head(results))

# by id, with a different formula
expr_by_id <- group_by(expr100, id)
results_by_tissue <- do(expr_by_id,
        subdf_to_pvals_df(., form = expression ~ tissue))
print(head(results_by_tissue))
```

July 10, 2015

The `write.table()` function can save a data frame to a text file.

```
write.table(expr,
            file = "../../data/expr_long_2_fixed.txt",
            quote = FALSE,
            row.names = FALSE,
            sep = "\t")
```

Then, in a new script we can load the file for analysis:

```
rm(list = ls())

expr <- read.table("../../data/expr_long_2_fixed.txt",
                   header = TRUE,
                   stringsAsFactors = FALSE,
                   sep = "\t",
                   comment.char = "")

print(head(expr, n = 50))
```

First things first: extracting data for a single gene:

```
first_id <- unique(expr$id)[1]
expr1 <- expr[expr$id %in% first_id, ]
```

Linear model:

```
lm1 <- lm(expr1$expression ~ expr1$genotype + expr1$treatment +
            expr1$genotype:expr1$treatment)

lm1 <- lm(expression ~ genotype + treatment + genotype:treatment, data = expr1)
print(lm1)
str(lm1)
```

What the heck is a "formula"? A container for variable names and how we want them to relate to each other.

```
# an aside: formula
f <- expression ~ genotype + treatment + genotype:treatment
print(f)
print(class(f))
print(unclass(f))

f <- alpha ~ delta + beta:gamma
print(f)
print(all.vars(f))


## back
```

Now the ANOVA based on the linear model, which is a list and a data frame:

```
anova1 <- anova(lm1)
print(anova1)
str(anova1)
```

We want to create a single-row data frame of pvalues. Try # 1, hard coding the column names:

```
pvals1 <- anova1$"Pr(>F)"
pvals1_df <- data.frame(pvals1[1], pvals1[2], pvals1[3])
colnames(pvals1_df) <- c("genotype", "treatment", "genotype:treatment")
print(pvals1_df)
```

Try # 2, wherein we get the column names from the rownames of the `anova1` dataframe. We gotta be careful though; `data.frame()`, when given a single vector, will create a data frame with a single column containing that vector. But, when given a list, will create columns for each element of the list (since data frames are a type of list).

```
pvals1 <- anova1$"Pr(>F)"
# convert the vector to a list to have the elements show up as columns
pvals1 <- as.list(pvals1)
pvals1_df <- data.frame(pvals1)
colnames(pvals1_df) <- rownames(anova1)
print(pvals1_df)
```

In fact, this operation is a good one to build a function out of:

```
# given a data frame with columns for genotype and treatement and expression,
# returns a dataframe of 3 pvalues, for geno, treat, and interaction
subdf_to_pvals_df <- function(subdf) {
  lm1 <- lm(expression ~ genotype + treatment + genotype:treatment,
            data = subdf)
  anova1 <- anova(lm1)

  pvals1 <- anova1$"Pr(>F)"
  pvals1 <- as.list(pvals1)
  pvals1_df <- data.frame(pvals1)
  colnames(pvals1_df) <- rownames(anova1)
  return(pvals1_df)
}
```

Which we can run quite easily:

```
res <- subdf_to_pvals_df(expr1)
print(res)
```

We wrote our own "interquartile range" function:

```
ipercentile_range <- function(x) {
  lower_val <- quantile(x, 0.25)
  upper_val <- quantile(x, 0.75)
  return(upper_val - lower_val)
}
```

And we explored the fact that *functions are a type of data*, just like any other:

```
print(class(ipercentile_range))
print(ipercentile_range)
```

Which means they can be passed as parameters to other functions. The boring way to run this function on 3 vectors inside of a list, collating the results into a list:

```
sample <- list()
sample$s1 <- rnorm(100, mean = 10, sd = 4)
sample$s2 <- rnorm(100, mean = 5, sd = 2)
sample$s3 <- rnorm(100, mean = 20, sd = 8)

# the boring way: run the function on each element individually
irq1 <- ipercentile_range(sample$s1)
irq2 <- ipercentile_range(sample$s2)
irq3 <- ipercentile_range(sample$s3)
sample_irqs <- list(irq1, irq2, irq3)
print(sample_irqs)
```

The cooler way, using the split-apply-combine strategy of `lapply()`:

```
sample_irqs <- lapply(sample, ipercentile_range)
print(sample_irqs)
```

Yeah.

July 9, 2015

This isn't a "class code" per se, but I just wanted to illustrate a problem I am currently dealing with in R, related directly to the concepts we've been talking about, but that I haven't figured out yet! You are welcome to try.

We have a data set, stored in `longdata`. We suspect that some of the groups of data are throwing off the results, so we'd like to try running the analysis with some of the data removed. Potential "bad" groups are stored in the data frame `bad`.

```
> print(head(longdata))
    Replicate Experiment Transformant Genotype Treatment Time Gene Expression
201         1         14            1      GFP      mock   24  ACH -0.5403274
202         1         13            1    VAM7m      mock   24  ACH -0.1602586
203         1         14            1     FYVE      mock   24  ACH -0.8098371
204         1         13            1    VAM7p      mock   24  ACH -0.3626171
205         1         14            1      GFP      BABA   24  ACH -1.1031427
206         1         13            1    VAM7m      BABA   24  ACH -0.1384794
> print(head(bad))
  Replicate Time Genotype
1         2    6     FYVE
2         6   12     FYVE
3         1   12     FYVE
4         3    0      GFP
5         3    1      GFP
6         4    3    VAM7m
```

So, I need to remove rows from `longdata` according to the criteria in `bad` (as in, any row with rep 2 and time 6 and genotype FYVE should be removed, according to the first row of `bad`). How can I do it? (Ah, I just figured out one possibility in posting it; using the `str_c()` function from the `stringr` library. But I wonder if it's possible using just the tools we've discussed in class thus far?)

July 8, 2015

Where we had left off from last time:

```
rm(list = ls())
library(stringr)

expr <- read.table("../../data/annon_2_long_annoying.txt",
                   header = TRUE,
                   sep = "\t",
                   comment.char = "#")
print(head(expr))

# split the sample col, set as df, set names, cbind to original
sample_split <- str_split_fixed(expr$sample, "_", 3)
sample_split_df <- data.frame(sample_split)
colnames(sample_split_df) <- c("genotype", "treatment", "tissuerep")
expr <- cbind(expr, sample_split_df)

print(head(expr))

# now, create a tissue column based on the presence of "A", "B", and "C"
# in the tissuerep column. Initialize it to NA, then check to see if any are
# still NA

expr$tissue <- NA
a_detect <- str_detect(expr$tissuerep, "A") # logical vector
expr$tissue[a_detect] <- "A"
expr$tissue[str_detect(expr$tissuerep, "B")] <- "B"
expr$tissue[str_detect(expr$tissuerep, "C")] <- "C"
print(expr[is.na(expr$tissue), ])

print(head(expr))
```

Then we created a `rep` column, but found that some rows still had `NA` after assignment of replicate values:

```
expr$rep <- NA
expr$rep[str_detect(expr$tissuerep, "1")] <- "1"
expr$rep[str_detect(expr$tissuerep, "2")] <- "2"
expr$rep[str_detect(expr$tissuerep, "3")] <- "3"
print(expr[is.na(expr$rep), ])
```

So we briefly discussed the `%in%` operator:

```
a <- c(3, 2, 5, 1) %in% c(1, 2)  # a has FALSE TRUE FALSE TRUE
```

And used it to remove all rows with `id` matching those with `NA` in the `rep` column. (IE, remove all data for genes with messy replicate data.)

```
bad_ids <- expr$id[is.na(expr$rep)]
bad_rows <- expr$id %in% bad_ids

expr <- expr[!bad_rows, ]
print(expr[is.na(expr$rep), ])
print(head(expr))
```

We discussed factors; trying to `str()` them and print their `class()`.

```
treat_fac <- factor(expr$treatment)
print(head(treat_fac))

str(treat_fac)
print(class(treat_fac))
```

Class is an attribute; there are special accessor functions for it though. By removing this attribute we can inspect the contents more clearly:

```
print(attr(treat_fac, "class"))
attr(treat_fac, "class") <- NULL
# preferred shortcut:
treat_fac <- unclass(treat_fac)
str(treat_fac)

attr(treat_fac, "class") <- "factor"
# preffered shortcut:
class(treat_fac) <- "factor"
```

Working with the levels attribute for renaming:

```
print(head(treat_fac))
attr(treat_fac, "levels") <- c("Chemical", "Water")
# preferred shortcut:
levels(treat_fac) <- c("Chemical", "Water")
print(head(treat_fac))
```

Reordering levels:

```
# reordering:
expr$treatment <- factor(expr$treatment,
                         levels = c("control", "chemical"),
                         ordered = TRUE)

# reordering and renaming simultaneously:
expr$treatment <- factor(expr$treatment,
                         levels = c("control", "chemical"),
                         labels = c("Water", "Chemical"),
                         ordered = TRUE)

# reodering computationally (in this case in reverse alphabetic order)
levs <- unique(expr$treatment)
levs_sorted <- levs[order(levs)]
levs_rev <- rev(levs)
expr$treatment = factor(expr$treatment,
                        levels = levs_rev,
                        ordered = TRUE)
```

Reordering using `reorder()`:

```
species <- c("salmon", "trout", "salmon", "bass", "bass")
weights <- c(10.2, 3.5, 12.6, 6.2, 6.7)

species_fac <- reorder(species, weights, mean)


# an example using a dataframe; order diamond colors
# by standard deviation of price, plot a boxplot
library(ggplot2)

diamonds$color <- reorder(diamonds$color, diamonds$price, sd)
p <- ggplot(diamonds) +
  geom_boxplot(aes(x = color, y = price))

plot(p)
```

July 6, 2015

A bit about data frames, their rownames and column names:

```
rm(list = ls())

id <- c("AGP", "T34", "ALQ", "IXL")
len <- c(256, 134, 92, 421)
gc <- c(0.21, 0.34, 0.41, 0.56)

genes <- data.frame(id, len, gc)

print(names(genes))
print(colnames(genes))

colnames(genes) <- c("geneid", "length", "gc")
```

Using them like lists (of columns)

```
subframe <- genes[c(1,3)]
subframe <- genes[c("geneid", "gc")]
subframe <- genes[c(TRUE, FALSE, TRUE)]


ids_vec <- genes[[1]]
# or (and this is better, since it doesn't depend on column order):
ids_vec <- genes[["geneid"]]
# or
ids_vec <- genes$geneid
# or
colname_extract <- "geneid"
ids_vec <- genes[[colname_extract]]


# if we wanted delete the length column:
# genes$length <- NULL
```

Row and column names are always character vectors, even if the quotes are left off in printing. A special `[row_selector, col_selector]` syntax:

```
print(rownames(genes))
print(genes)

subframe <- genes[c(3, 1), c("length", "geneid")]
print(subframe)

rownames(genes) <- c("g1", "g2", "g3", "g4")
subframe <- genes[c(3, 1), c("length", "geneid")]
print(subframe)
```

Some dataframe operations examples in practice, supporting a variety of ways of organizing, chaining, and nesting the operations.

```
row_log_selector <- genes$length < 200 | genes$gc < 0.3
shorties <- genes[row_log_selector, ]
# or
shorties <- genes[genes$length < 200 | genes$gc < 0.3, ]
# or (subset uses "non standard evaluation")
shorties <- subset(genes, length < 200 | gc < 0.3)


shorties_gc_vec <- shorties$gc
# or
shorties_gc_vec <- genes[row_log_selector, ]$gc
# or
shorties_gc_vec <- genes$gc[row_log_selector]


genes_sorted <- genes[order(genes$length), ]
```

Adding columns by assigning to existing names, and selective replacement within columns (also see HW1)

```
genes$gc_category <- NA
genes$gc_category[genes$gc < 0.5] <- "low"
genes$gc_category[genes$gc >= 0.5] <- "high"
# any still left as NA?
print(genes[is.na(genes$gc), ])
```

Working with annoying character data columns:

```
expr <- read.table("~/Downloads/annon_2_long_annoying.txt",
                   header = TRUE,
                   sep = "\t",
                   comment.char = "#")

print(head(expr))


library(stringr)
sample_split <- str_split_fixed(expr$sample, "_", 3)
print(head(sample_split))

sample_split_df <- data.frame(sample_split)
colnames(sample_split_df) <- c("genotype", "treatment", "tissuerep")
print(head(sample_split_df))

# be careful with cbind....
expr <- cbind(expr, sample_split_df)


expr$tissue <- NA
a_detect <- str_detect(expr$tissuerep, "A")
expr$tissue[a_detect] <- "A"

expr$tissue[str_detect(expr$tissuerep, "B")] <- "B"
expr$tissue[str_detect(expr$tissuerep, "C")] <- "C"

print(expr[is.na(expr$tissue), ])

print(head(expr))
```

The large expression file can be downloaded **here**.

July 1, 2015

We started by looking at lists, and extracting sublists by index vector and logical vector:

```
rm(list = ls())

organism <- "A thaliana"
ecotypes <- c("C24", "Col0", "WS2")
numchrs <- 5

athal <- list(organism, ecotypes, numchrs)
print(class(athal))      # prints "list"

sublist <- athal[c(1,3)]                # list with two elements
sublist <- athal[c(TRUE, FALSE, TRUE)]  # same
```

Next up, using `[]` returns a sublist of length 1. We need to use `[[]]` syntax to extract the element itself from inside the list. The results of `print()` reflect the `[[]]` syntax for the locations of the elements.

```
ecotypes <- athal[2]                    # ahah- a list of length 1
print(length(ecotypes))
print(class(ecotypes))

ecotypes_vec <- athal[[2]]              # the vector inside the list
print(ecotypes_vec)

second_ecotype <- athal[[2]][2]         # chaining []-syntax of various types

print(athal)
```

Lists are commonly named and accessed by name vector:

```
names(athal) <- c("organism", "ecotypes", "numchrs")
sublist <- athal[c("organism", "numchrs")]
ecotypes_vec <- athal[["ecotypes"]]
# same as
ecotypes_vec <- athal$"ecotypes"
# same as (if the name contains no funky characters)
ecotypes_vec <- athal$ecotypes
```

If the name we want to extract is stored in a variable, we have to use `[[]]` rather than `$`.

```
## extracting by name stored in variable
extract_name <- "ecotypes"
ecotypes_vec <- athal[[extract_name]]
## note that we cannot use $ for this due to the way it works
```

We can create an empty list and assign to elements directly by name:

```
# another list
chrs <- list()
chrs$lengths <- c(54.2, 36.4, 19.7, 46.3, 29.2)
chrs$genes <- c(6842, 2355, 1366, 1515, 1666)
```

Lists can contain other lists. We also talked briefly about attributes, which can be associated with any data element. The `str()` function is a nicer way to print the "structure" of a list.

```
# list within lists! and attributes
athal$chromosomes <- chrs
attr(athal$organism, "kingdom") <- "Plantae"
print(athal)
str(athal)       # str: a nicer way to look at lists
```

Deleting elements from a list, or an attribute, can be accomplished by setting it to `NULL` with `<-`.

```
athal$ecotypes <- NULL    # deleting a list element or an attribute
str(athal)
```

Lists of lists with annotations, etc. are important because most statistical functions in R return lists. Even though `print()` reveals a nicely formatted output, `str()` shows the structure of the list. And we can use that information to extract individual pieces of information (like p-values).

```
samp1 <- rnorm(100, mean = 20, sd = 10)
samp2 <- rnorm(100, mean = 15, sd = 10)
tres <- t.test(samp1, samp2)
print(tres)
str(tres)

pval <- tres$p.value # or tres[["p.value"]]
print(pval)
```

Perhaps that's the kind of thing we'd like to wrap up in a function.

```
ttest_pval <- function(samp1, samp2) {
  tres <- t.test(samp1, samp2)
  pval <- tres$p.value
  return(pval)
}


a <- rnorm(100, mean = 20, sd = 10)
b <- rnorm(100, mean = 15, sd = 10)
print(ttest_pval(a, b))
```

And we finally briefly covered if-statements, which we won't return to again for a week or two.

```
a <- 5
# (I probably should have use >s in this example in class)
if(a > 100) {
  print("fyi:")
  print("a is greater than 100")
} else if(a > 50) {
  print("a is greater than fifty but less than 100")
} else if(a > 25) {
  print("a is greater than 25 but less than 50")
} else {
  print("a is less than or equal to 25")
}


a <- 5
if(a < 100) {
  print("a is less than 100")
} else {
  print("a is greater than or equal to 100")
}
```

June 29, 2015

We explored a caveat relating to R's printing of numerics with many integer places:

```
#clear environment
rm(list = ls())

# printing big numbers...
a <- 111111.98
print(a)
print(as.integer(111111.98))
print(a, digits = 10)
```

We wrote a function!

```
# a wild function appears!
equalish <- function(a, b, epsilon = 0.00001) {
  result <- abs(a - b) < epsilon
  return(result)
}

x <- c(4, 2)
y <- c(4.000000001, 6)
res <- equalish(x, y)
# alternate ways to call
# by position only
res <- equalish(x, y, 0.00001)
# by name only
res <- equalish(a = x, b = y, epsilon = 0.00001)
# by mix (do by positions first, then by names)
res <- equalish(x, y, epsilon = 0.00001)
```

We explored some properties of local variables in functions

```
# a test of local variables
equalish <- function(a, b, epsilon = 0.00001) {
  result <- abs(a - b) < epsilon
  testvar <- 100  # local variable
  return(result)
}

testvar <- "hellooooo"
x <- c(4, 2)
y <- c(4.000000001, 6)
res <- equalish(x, y)
print(res)
print(testvar)   # prints "helloooo"
# print(b)         would be an error (b is a local variable too)
```

And some properties of non-local variables:

```
# a different test
equalish <- function(a, b, epsilon = 0.00001) {
  result <- abs(a - b) < epsilon
  print(testvar)
  return(result)
}

testvar <- "hellooooo"
x <- c(4, 2)
y <- c(4.000000001, 6)
res <- equalish(x, y)
```

Which means that we could, theoretically, not use parameters. Though this is a bad idea and breaks "rule 1":

```
# baaaad
equalish <- function() {
  result <- abs(a - b) < epsilon
  return(result)
}

a <- c(4, 2)
b <- c(4.00000001, 6)
epsilon <- 0.00001
res <- equalish()
```

And we looked at reading help documentation etc.

```
# help stuff
help("read.table")
?read.table
help.search("average")
??average

# load a library, get some help on it
library(ggplot2)
help(package = "ggplot2")
```

June 26, 2015

Some basic usage of data frames:

```
# clear environment
rm(list = ls())

states <- read.table(file = "/Users/soneil/states.txt",
                     header = TRUE,
                     sep = "\t",
                     stringsAsFactors = FALSE,
                     comment.char = "#")
print(class(states))
print(states)

first_10 <- head(states, n = 10)
print(first_10)

num_rows <- nrow(states)  # integer 50
num_cols <- ncol(states)  # integer 6

incomes <- states$"income"
incomes <- states$income
print(class(incomes))  # numeric
print(incomes)

# a quick test...
names <- c("Jerry", "Kramer")
ages <- c(30, 34, 25, 37)
seinfeld <- data.frame(ages, names)
```

Selection and selective replacement by index # vector:

```
# selection by index vector
nums <- c(10, 20, 30, 40)
second_el <- nums[2]
# same as
second_sl <- nums[c(2)]

subvector <- nums[c(3,2)]
print(subvector)     # 30 20

# selective replacement
nums[c(3, 2)] <- c(35, 25)
print(nums)          # 10 25 35 40
```

By name vector:

```
scores <- c(89, 94, 73)
names(scores) <- c("Student A", "Student B", "Student C")

print(scores)  # prints the vector and the names
names_vec <- names(scores)   # character vector of "Student A" "Student B" "Stude

ca_scores <- scores[c("Student C", "Student A")]    # named vector 73 89

scores[c("Student C", "Student A")] <- c(75, 92)
print(scores)    # named vector, 92 94 75
```

And by logical vector:

```
scores <- c(92, 94, 73)
select_vec <- c(TRUE, FALSE, TRUE)
scores_subvec <- scores[select_vec]   # 2-element vector, 92 73

# replacement by logical
scores[select_vec] <- c(96, 85)
print(scores)    # 96 94 85
```

Operations and functions are "vectorized"

```
nums_strs <- c("6", "3.7", "9b3x")
nums <- as.numeric(nums_strs)
print(nums)   # includes an NA

na_test <- NA + 3
print(na_test)         # prints NA
print(class(na_test))  # numeric

values <- c(10, 20, 30, 40)
mult <- c(1, 2, 3, 4)
result <- values * mult  # 10 40 90 160

comp_values <- c(25, 10, 35, 25)
result <- values > comp_values # FALSE TRUE FALSE TRUE
```

Vector recycling:

```
# vector recycling
values <- c(10, 20, 30, 40)
mult <- c(1, -1)
result <- values * mult # 10 -20 30 -40

result <- values * 2 # same as values * c(2); resulting in 20 40 60 80

# partial recycling
values <- c(3, 5, 7)
mult <- c(10, -10)
result <- values * mult  # 30 -50 70 (and and warning)

# recycling and selective replacement
values <- c(10, 20, 30, 40, 50, 60)
values[c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)] <- c(5, -5)
print(values)      # 5 -5 30 5 50 -5

values[c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)] <- 0
print(values)      # 0 0 30 0 50 0
```

Combining these features for powerful data manipulation:

```
values <- c(50, 10, 60, 20, 40, 30)
select_vec <- values > 35      # recycling to produce T F T F T F
gt_35 <- values[select_vec]    # 50 60 40
# OR
gt_35 <- values[values > 35]

# we can also use this technique to selectively replace elements with NA
values[values > 35] <- NA

## Similarly, we can use the mean() function rather than hard-coding 35
values <- c(50, 10, 60, 20, 40, 30)
gt_mean <- values[values > mean(values)] # 50 60 40
# and we can replace all values larger than the mean with a value, like NA
values[values > mean(values)] <- NA
```

June 24, 2015

Character data type:

```
# clear environment
rm(list = ls())

first <- 'Shawn'
last <- "O'Neil"
full <- 'Shawn\tO\'Neil'

var_char <- "0.12"
var_num <- 0.4
# sum <- var_char + var_num            # would be an error
var_char_converted <- as.numeric(var_char)
sum <- var_num + var_char_converted    # 0.52
```

Logicals:

```
sun_is_yellow <- TRUE
test <- 3 < 5          # TRUE

testchar <- "alice" < "bob"         # TRUE
testnonsense <- "alice" < 10000000  # TRUE (ugh)

# watch out: these have different meanings
a <- 5
a < -5
a = -5
a == -5

a <- TRUE
b <- FALSE

c <- a | b        # TRUE
c <- a & b        # FALSE
c <- !a           # FALSE

a <- "alice"
b <- "bob"
c <- 3
d <- 4.7
test <- c < d & (b == "bob" | !(a < b))    # TRUE
```

Beware of testing equality (and inequality) of numerics that have decimal places:

```
a <- 0.2
b <- 0.2 * 0.2 / 0.2
print(a == b)             # prints FALSE!

epsilon <- 0.0000001
test <- abs(a - b) < epsilon
print(test)               # prints TRUE
```

Vectors!

```
vec1 <- c(3.2, 4.7, -3.5)  # 3-element vector
vec2 <- c(20.4, vec1, 37.6) # 5-element vector
first_el <- vec2[1]         # 20.4
second_el <- vec2[2]        # 3.2

num_els <- length(vec2)     # integer 5
last_el <- vec2[num_els]    # 37.6
# OR
last_el <- vec2[length(vec2)]  # 37.6
```

"No naked data" and "vectors have (a) class"

```
# a vector of length 1
age <- 21
# same as:
age <- c(21)

range <- seq(1, 20, 0.5)
print(range)

range <- seq(1, 20, 1)
range <- 1:20

print("hello")

# autoconversion of mixed types in a vector
test <- c(TRUE, FALSE, as.integer(20))
print(test)
print(class(test))

test <- c(TRUE, FALSE, as.integer(20), 3.7)
print(test)
print(class(test))

test <- c(TRUE, FALSE, as.integer(20), 3.7, "hi")
print(test)
print(class(test))
```

Breaking lines and read.table:

```
# ok
total <- 4 + 5 + 6 + 7

# ok
total <- 4 + 5 +
         6 + 7

# not ok
total <- 4 + 5
         + 6 + 7



states <- read.table(file = "/Users/soneil/states.txt",
                     header = TRUE,
                     sep = "\t",
                     stringsAsFactors = FALSE,
                     comment.char = "#")
```

June 22, 2015

Some basics of variables and printing and such:

```
rm(list = ls())   # removes/clears all variables

message <- "hello"
print(message)

alpha <- -4.4
print(alpha)

alpha_abs <- abs(-4.4)
alpha_abs <- abs(alpha)
print(alpha_abs)

alpha <- -6.6
alpha <- abs(alpha)
print(alpha)

alpha_int <- as.integer(alpha)   # integer 6
print(class(alpha_int))
print(class(alpha))

alpha_re_numeric <- as.numeric(alpha_int)
print(class(alpha_re_numeric))  # numeric 6.0

## other variable names and assignment types we won't use:
alpha.abs <- abs(alpha)
alpha_abs = abs(alpha)
`absolute value of alpha` <- abs(alpha)
```

June 17, 2015

Here's where I'll be posting code that I write on the board in class.

You should still take notes though.

http://www.scientificamerican.com/article/a-learning-secret-don-t-take-notes-with-a-laptop/

http://www.scientificamerican.com/article/reading-paper-screens/