Practicum #2

Due: Tuesday, November 9 @ 6pm

Adapted from Dr. Rosemary Braun

Part I: Loading and exploring the data

In this tutorial/homework, we'll be working with microarray data from:

> Chiaretti et al (2004), Gene expression profile of adult T-cell acute lymphocytic leukemia identifies distinct subsets of patients with different response to therapy and survival, Blood 103:7.

NOTE: The data we'll use today has already been preprocessed for analysis. Preprocessing of microarray and next generation sequencing data is an important step that is beyond the scope of this class, but you should be aware that this is something you cannot overlook when working with data of your own!

Follow these steps to load the data:

- Download from Canvas (or GitHub) the file `all_genex_data.RData` and save it somewhere convenient on your computer. (For demonstration purposes, I'll save mine to the desktop of my Mac; yours may be different.) This file is an R Data file, with the data stored in a binary format that R likes. You need to load it into R to look at and use it (you can't load it in excel), which we'll do in RStudio – don't double-click on it yet!
- Open RStudio and clear your workspace using the `Session` menu or the broom icon in the `Environment` pane on the upper left. This ensures that you'll start fresh. Also, open a new R Script file (or markdown) where you can save your work.
- Find your saved `all_genex_data.RData` file in the "Files" pane of RStudio (should be in the bottom left) and double-click on it there. RStudio will ask you if you want to load it; tell it yes.
- At this point, your Environment pane should show three things `ALL.expr`, `ALL.pheno`, and `keggPathways`. If not, raise your hand & we'll help you out!
- If loading the data was successful, you'll see a line that looks similar to the following in your R Console window:

```
load("~/Desktop/all_genex_data.RData")
```

(NOTE: yours may be a bit different!) Cut & paste this as the first line of your R Script. That way, every time you run your script it'll automatically load the data first. Be sure to add a comment!

Check out the data structure

`phenotype_df` - phenotype data for the subjects in the study

A data frame is rectangular: every column has the same number of rows, and every row has the same number of columns. You can access entries in it using `[row, column]` and you can also access the columns with `$`. One useful thing about a dataframe is that the columns can be of different types; some columns can be numeric, while others may be character strings or boolean.The ability to mix different data types is useful when you want to tabulate things that aren't all the same type for a common set of samples (e.g., if I wanted to store a table of the (numeric) weights and (string) breeds of all the dogs in an animal shelter).

1. **Read the help pages for `dim`, `nrow`, and `ncol`. Using one or more of these functions, find out how many rows and columns the `phenotype_df` dataframe has.**

We can access elements of a dataframe using `data[rows,columns]`:

```
# select row 1, column 1
phenotype_df[1,1]
```

```
## [1] "s01005"
```

```
# select rows 1:5, all columns
phenotype_df[1:5,]
```

```
##   subject_id  cod diagnosis sex age BT remission CR   date.cr t(4;11) t(9;22)
## 1     s01005 1005 5/21/1997   M  53  B        CR CR  8/6/1997   FALSE    TRUE
## 2     s01010 1010 3/29/2000   M  19  B        CR CR 6/27/2000   FALSE   FALSE
## 3     s03002 3002 6/24/1998   F  52  B        CR CR 8/17/1998      NA      NA
## 4     s04006 4006 7/17/1997   M  38  B        CR CR  9/8/1997    TRUE   FALSE
## 5     s04007 4007 7/22/1997   M  57  B        CR CR 9/17/1997   FALSE   FALSE
##   cyto.normal       citog mol.biol fusion protein mdr   kinet   ccr relapse
## 1       FALSE     t(9;22)  BCR/ABL           p210 NEG dyploid FALSE   FALSE
## 2       FALSE simple alt.      NEG           <NA> POS dyploid FALSE    TRUE
## 3          NA        <NA>  BCR/ABL           p190 NEG dyploid FALSE    TRUE
## 4       FALSE     t(4;11) ALL1/AF4           <NA> NEG dyploid FALSE    TRUE
## 5       FALSE     del(6q)      NEG           <NA> NEG dyploid FALSE    TRUE
##   transplant               f.u date last seen BTstage stage
## 1       TRUE BMT / DEATH IN CR           <NA>      B2     2
## 2      FALSE               REL      8/28/2000      B2     2
## 3      FALSE               REL     10/15/1999      B4     4
## 4      FALSE               REL      1/23/1998      B1     1
## 5      FALSE               REL      11/4/1997      B2     2
```

```
# we could get the ages for the first 5 like this:
phenotype_df[1:5,"age"]
```

```
## [1] 53 19 52 38 57
```

```
# or like this, using $
phenotype_df$age[1:5]
```

```
## [1] 53 19 52 38 57
```

We can also filter our dataframe using `dplyr::filter()`

> Note: Make sure you have loaded the `tidyverse` package at the top of your script first using `library(tidyverse)`

```
# filter to keep only females
females <- dplyr::filter(phenotype_df, sex == "F")

# we can also filter for several things at once:
old_females <- dplyr::filter(phenotype_df, sex == "F", age > 50)
```

Take a quick look at each of the columns, what type of data are we looking at? What are the unique values or the distribution of the values for each column? (*Hint: check out `?unique`*)

`expression_df` - expression data for many genes across 128 subjects

Notice this dataframe has a column for each subject and a row for each gene. This data is in what we call "wide" format. We can convert it into "long" format using the function `pivot_longer()` from the `tidyr` package. Try it out with the following command:

```
long_expr <- tidyr::pivot_longer(expression_df, cols = -gene, names_to = "subject_id", values_to = "expres
```

Compare the two dataframes. They both contain the same information, but structured in a different way. As a data scientist, we might want the "wide" format for some types of calculations and we might want the "long" format for others, therefore it is important to know how to switch between them. For practice, try converting the `long_expr` dataframe back to a wide dataframe by running the following code:

```
wide_expr <- tidyr::pivot_wider(long_expr, names_from = "subject_id", values_from = "expression")
```

> Notice that the `wide_expr` and the original `expression_df` dataframes are the exact same? Also notice the syntax for the `pivot_*()` functions - in `pivot_longer()` we are creating two new columns from all the data (we are *gathering* all the columns into a **key** and **value** pair). Alternatively, in `pivot_wider()` we are taking those two columns and *spreading* them back out again, so each row in the `names_from` column will become a new column with the value from `values_from` column.

2. **Let's practice filtering and plotting the data:**

    a. **Plot a histogram of the expression of MAPK3 in all samples.**
    b. **Using `plot()`, make a scatterplot of the expression of DUSP1 vs TP53.**

We might also find it useful to join two datasets together by a common variable (i.e. column).

```
merged <- dplyr::full_join(long_expr, phenotype_df, by = "subject_id")
```

> Note: there are different types of joining (check out ?join for more). `dplyr::full_join()` will ensure that all rows in both the data frames will be kept in a merge. Notice how the rows in the `phenotype_df` dataframe have been copied for each subject-gene pair.

`kegg_pathways` - a list of KEGG pathways and each of the genes that belong to each pathway

Lists are another special type of R data structure. To learn more about lists, let's first create a new (smaller) list to practice on.

```
x <- list(a = 1:4, b = "hi", c = FALSE)
x
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] "hi"
##
## $c
## [1] FALSE
```

`x` is a list of 3 items: "a", "b", and "c". Each item is associated with a vector (i.e. 1:4, "hi", and `FALSE`). To extract the value of an item from a list, you can use the `$`:

```
# extract the value from item "a" in list x
x$a
```

```
## [1] 1 2 3 4
```

If we don't know the list item names, we can use the `names()` command:

```
names(x)
```

```
## [1] "a" "b" "c"
```

We can also refer to items in a list by the index number, similar to how we can subset vectors. For example, the following code will extract the third element of list `x`:

```
x[[3]]
```

```
## [1] FALSE
```

> Note the use of **double brackets** [[ ]] instead of **single brackets** [ ] like we use for a vector call. For a bonus, try to extract the third element of the first item in list `x`.

Let's put this to use. The R function `t.test()` outputs a list. We could learn about it from the help page under the "Value" section, or we can learn by poking at it. Let's try:

```
# 10 random #'s from a std normal
samp1 <- rnorm(10)

# 20 random #'s from a std normal
samp2 <- rnorm(20)

# t-test of the above; store the output in null.t.out null.t.out
null.t.out <- t.test(samp1,samp2)

# look at the output
null.t.out
```

```
##
##  Welch Two Sample t-test
##
## data:  samp1 and samp2
## t = -0.54456, df = 20.841, p-value = 0.5918
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.9679931  0.5663935
## sample estimates:
## mean of x mean of y
## 0.1338165 0.3346163
```

Check out the structure of the `t.test()` output with `str(null.t.out)`. You should see that it is a list of 10 items! You can also see the name of each item of the list and the type of values each item contains.

```
str(null.t.out)
```

```
## List of 10
##  $ statistic  : Named num -0.545
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 20.8
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 0.592
##  $ conf.int   : num [1:2] -0.968 0.566
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num [1:2] 0.134 0.335
##   ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "difference in means"
##  $ stderr     : num 0.369
##  $ alternative: chr "two.sided"
##  $ method     : chr "Welch Two Sample t-test"
##  $ data.name  : chr "samp1 and samp2"
##  - attr(*, "class")= chr "htest"
```

Let's try extracting the p-value from the `t.test()` output:

```
null.t.out$p.value
```

```
## [1] 0.5918379
```

3. **Look at the help page for t.test:**

    a. **Write an expression to extract the degrees of freedom from `null.t.out`**
    b. **Write an expression to extract the upper bound of the confidence interval from `null.t.out`**

Let's go back to our `kegg_pathways` list and take a look at the first three entries.

```
kegg_pathways[1:3]
```

```
## $`Complement and coagulation cascades`
##  [1] "PLAUR"    "SERPIND1" "C3AR1"    "SERPINF2" "CFH"      "KLKB1"
##  [7] "MBL2"     "C2"       "MASP1"    "PLAT"     "F7"       "BDKRB1"
## [13] "THBD"     "F13B"     "F5"       "C8G"      "C9"       "F9"
## [19] "F11"      "CFI"      "PROS1"    "CFB"      "CR1"      "C8B"
## [25] "F3"       "FGG"      "PLG"      "SERPINA1" "CR2"      "SERPINA5"
## [31] "FGB"      "CPB2"     "SERPINC1" "F2"       "KNG1"     "PLAU"
## [37] "C7"       "MASP2"    "F8"       "C5"       "F12"      "F13A1"
## [43] "SERPINE1" "C4BPB"    "CD46"     "C1QB"     "FGA"      "CD59"
## [49] "PROC"     "BDKRB2"   "C1R"      "CD55"     "SERPING1" "F10"
## [55] "CFD"      "C1S"      "TFPI"     "C8A"      "C4BPA"    "F2R"
## [61] "C6"       "VWF"
##
## $`Caffeine metabolism`
## [1] "CYP1A2"  "CYP2A7"  "CYP2A6"  "CYP2A13" "XDH"     "NAT1"    "NAT2"
##
## $`Drug metabolism - other enzymes`
##  [1] "CDA"     "TK2"     "CYP2A7"  "CYP2A6"  "CYP2A13" "TYMP"    "CYP3A4"
##  [8] "UGT2B11" "TPMT"    "UGT2B15" "GUSB"    "XDH"     "UGT2B17" "UMPS"
```

```
## [15] "UCKL1"   "ITPA"    "GMPS"    "IMPDH2"  "UGT2B4"  "UPP1"    "HPRT1"
## [22] "NAT1"    "DPYD"    "CYP3A5"  "NAT2"    "IMPDH1"  "CES2"    "UGT2B7"
## [29] "TK1"     "DPYS"
```

We can see that keggPathways maps a KEGG pathway name to a list of genes. This is pretty handy! For instance, if I want the gene expression data for only the genes on the caffeine metabolism pathway, for the first three samples, it would be:

```
# gives me the names of the genes I want
kegg_pathways$"Caffeine metabolism"
```

```
## [1] "CYP1A2"  "CYP2A7"  "CYP2A6"  "CYP2A13" "XDH"     "NAT1"    "NAT2"
```

```
# now I can filter the dataframe to only include these genes
caffeine_exp <- dplyr::filter(wide_expr, gene %in% kegg_pathways$"Caffeine metabolism")
```

Note: we could have also filtered the ***`long_expr`*** *dataframe*

4. **Using `kegg_pathways`:**
    a. **How many pathways are in kegg_pathways? (*Hint: use* `length()`)**
    b. **How many genes are in the pathway called "p53 signaling pathway"?**
5. **Using what you have just learned, plot:**
    a. **A histogram of MAPK3 only in B-cell ALL patients, where `phenotype_df$BT=="B"`.**
    b. **A scatter plot of DUSP1 vs TP53 only in T-cell ALL patients, which is indicated by "T" in the BT columns of phenotype_df**

Part II: Testing for differential expression

Now we can start putting all these pieces together to look for differential expression—differences in the mean expression levels of certain genes in the T-cell and B-cell ALL samples.

Let's begin by testing just one gene, MAPK3, for differential expression. We'll do this by using phenotype_df$BT to separate out the B- and T-cell ALL cases:

```
# create a dataframe of expression for B-cell ALL
bcell <- dplyr::filter(merged, gene == "MAPK3", BT == "B")

# create a dataframe of expression for T-cell ALL
tcell <- dplyr::filter(merged, gene == "MAPK3", BT == "T")

# perform a t.test for expression
t.test(bcell$expression, tcell$expression)
```

```
##
##  Welch Two Sample t-test
##
## data:  bcell$expression and tcell$expression
## t = -3.9099, df = 60.675, p-value = 0.0002355
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.28299757 -0.09146783
## sample estimates:
## mean of x mean of y
##  7.499918  7.687151
```

6. **Modify the above to test whether MAPK3 expression differs in people younger than 20.**

Suppose we only wanted the p-value, not all the other stuff. How would we do it?

```
t.test(bcell$expression, tcell$expression)$p.value
```

```
## [1] 0.0002354671
```

7. **Modify the above to spit out TRUE when the expression is higher in B-cell ALL and FALSE when it's higher in T-cell ALL.**

So now what we'd like to do is repeat the above code for all the genes, and collect a p-value for each. Before we do that, though, there are some questions to consider:

8. **In the last block of code given above,**
    a. **What is the null hypothesis?**
    b. **What is the alternative hypothesis?**
    c. **If we were to do this test for all genes, how many tests would we do?**
    d. **Of those, how many would we expect to have $p < 0.05$ by pure chance?**

Looping over genes with "for"

There are a couple ways we can loop over all the genes to get p-values. The for loop is computationally slow, but easier to understand, so we'll start with it.

In a `for` loop, we give R a set of values to iterate a piece of code over. Here's an example in which I iterate over `1:4`, using its value in a `rep()` function:

```
for(i in 1:4){ # i gets set to 1, then 2, then 3, then 4
    print(rep("hi", i)) # i gets plugged in to this code
}
```

```
## [1] "hi"
## [1] "hi" "hi"
## [1] "hi" "hi" "hi"
## [1] "hi" "hi" "hi" "hi"
```

Here, we want to iterate the `t.test` over all the genes.

> Note: this will take a while to run! Be patient.

```
# First, we'll initialize a vector to hold our results.
gene.p.vals <- vector()

# note how similar this code is to the code above for MAPK3
for(gene_name in unique(long_expr$gene)) {
    bcell <- dplyr::filter(merged, gene == gene_name, BT == "B")
    tcell <- dplyr::filter(merged, gene == gene_name, BT == "T")
    gene.p.vals[gene_name] <- t.test(bcell$expression, tcell$expression)$p.value
}
```

9. **Explain what each line in the `for` loop does.**

Now suppose we wanted to look at only the p-values for the genes on the caffeine metabolism pathway. We could do that by subsetting `gene.p.vals`:

```
gene.p.vals[kegg_pathways$"Caffeine metabolism"]
```

```
##      CYP1A2      CYP2A7      CYP2A6     CYP2A13         XDH        NAT1
## 0.285070571 0.754014775 0.061856917 0.007379019 0.037709624 0.364796715
##        NAT2
## 0.818993677
```

Optional: looping over genes with `apply`

If you have used R before and are comfortable with writing for loops, check out this small section on `lapply`! Otherwise, feel free to skip this. You can always try it one day in the future.

Using the `lapply()` command is much more efficient (about 3x faster!) and a more "proper" way to use R. However, it can be a little confusing.

```
gene.p.vals2 <- sapply(unique(long_expr$gene), function(gene_name) {
    bcell <- dplyr::filter(merged, gene == gene_name, BT == "B")
    tcell <- dplyr::filter(merged, gene == gene_name, BT == "T")
    return(t.test(bcell$expression, tcell$expression)$p.value)
})
```

Notice that the syntax is different from that of the for-loop. Here, we had to create our own function inside the apply. That function was designed to take in a gene from the expression dataframe, store it in a vector called `gene_name`, and then extract the B- and T- cell values from that vector and compute the t-test.

If this is confusing to you, feel free to use the for loop instead for now!

Part III: Adjusting for multiple hypotheses

At this point, you should have a vector `gene.p.vals` of p-values for each gene.

10. **Using `gene.p.vals`,**
    a. **How many p values are less than 0.05? How many are less than 0.01? How do those compare to what you expected?**
    b. **Plot a histogram of the p-values. Does it look the way you'd expect? Why or why not?**
    c. **Use the `min()` function to find the smallest p-value.**
    d. **Use the `sort()` function to find the ten smallest p-values.**

In class, we discussed two types of corrections: the FWER "Bonferroni" correction pioneered by Olive Jean Dunn, and the more recent FDR correction pioneered by Benjamini & Hochberg.

11. **Compute the FWER (Dunn/Bonferroni) adjusted p-values, and store it as `gene.bonf.vals`. Then repeat parts (a),(c),(d) of the previous exercise using the adjusted p-vals.**
12. **Compute the FDR adjusted p-values, and store it as `gene.FDR.vals`. (Hint: use the `p.adjust` function.) Then repeat parts (a),(c),(d) of the previous exercise using the FDR-vals.**

Criteria for differentially expressed genes

You'll notice that a great number of genes pass the FDR < 0.01 threshold, even after adjustment. In order to whittle this down further, a common criteria for calling a "gene differentially expressed" is not only that it is significantly different in the two phenotypes; we also want a large difference in the gene expression. Often, we want it to be twice as big (or half as small) in one or the other group.

Preprocessed gene expression data is often log_2 transformed, and the data we are working with now is no exception (this has already been done). Working with log_2 transformed data is convenient, because it turns ratios (doublings and halvings) into differences. Suppose X_B is my original gene expression data in B-cell ALL, and X_T is my original gene expression data in T-cell ALL. Suppose further that $Y_B = log_2(X_B)$ and $Y_T = log_2(X_T)$. If we want to find genes that meet the criterion $X_T \geq 2X_B$ (i.e. genes with twice the expression in X_T as in X_B), we can use the fact that $log(AB) = log(A) + log(B)$ to derive (algebra not included):

$$Y_T - Y_B \geq 1$$

Similarly, if we want genes that have less than half the expression in X_T as in X_B, we can use the same algebra to derive

$$Y_T - Y_B \leq -1$$

or

$$Y_B - Y_T \geq 1$$

Putting these together, the doubling/halving criterion is

$$|Y_B - Y_T| \geq 1$$

Let's write a line that will spit out a single fold change, this time for the MAPK3 gene, in a similar way to our p-value code on page X (right before problem 7).

```
bcell <- dplyr::filter(merged, gene == "MAPK3", BT == "B")
tcell <- dplyr::filter(merged, gene == "MAPK3", BT == "T")

test_est <- t.test(bcell$expression, tcell$expression)$estimate
fold_change <- test_est[1]-test_est[2]
fold_change
```

```
##  mean of x
## -0.1872327
```

13. **What do the expressions `t.test(bcell$expression, tcell$expression)$estimate` and `test_est[1]-test_est[2]` do? Hint: look at `test_est`**

14. **Refer back to how we extended the p-value computation to loop over all genes.**

    a. **Create some code to compute the fold changes for all the genes and store them in a vector called `gene_fcs`. You may use either `for` or `apply` to do this.**
    b. **Now that you have `gene_fcs`, you can pick out the ones that meet the doubling/halving criteria with `which(abs(gene_fcs)>1)`. How would you modify this to get only the genes that are up-regulated in T-cell ALL?**
    c. **How many genes have an absolute fold change greater than 1.5?**

Putting this all together, let's call genes that have an absolute fold change greater than 1.5 and an FDR < 0.01 "differentially expressed." We'll combine these criteria with the boolean "and" operator, `&`:

```r
DEgenes <- ((abs(gene_fcs) > 1.5) & (gene.FDR.vals < 0.01))

# Note: DEgenes is a boolean (TRUE if the gene meets the DE criteria, FALSE otherwise) # To get the names

names(which(DEgenes))
```

```
##  [1] "FLT3"     "CD19"     "PTPRE"    "LCK"      "ITK"      "ZAP70"
##  [7] "CDKN1A"   "CD24"     "TCF7"     "LILRA2"   "CHI3L2"   "SH2D1A"
## [13] "LMO2"     "FHL1"     "GSN"      "TRBC1"    "TRAT1"    "TUBA4A"
## [19] "TOX"      "SIK1"     "PDE4B"    "GNAI1"    "IGHM"     "SLC2A5"
## [25] "CD1B"     "CD74"     "NUCB2"    "HLA-DQB1" "POU2AF1"  "CTGF"
## [31] "PDLIM1"   "MLLT11"   "JCHAIN"   "HLA-DRA"  "CD247"    "HLA-DMA"
## [37] "AKR1C3"   "HLA-F"    "LAPTM5"   "CD79B"    "CD79A"    "HBEGF"
## [43] "MAL"      "HLA-DPB1" "BLNK"     "CD3D"     "DSTN"     "TSPAN7"
## [49] "ZNF529"   "NPY"      "NOTCH3"   "HLA-DPA1" "SOCS2"    "CMAHP"
## [55] "TCL1A"    "CD9"      "NREP"     "YBX3"     "NRIP1"    "KLF9"
## [61] "GATA3"    "FOXO1"    "TFPI"     "ITM2A"    "CTBP2"    "CRIM1"
## [67] "THEMIS2"  "HLA-DMB"  "HLX"
```

15. **How many genes are differentially expressed? (Use R, don't just count them off the page!)**

Part IV: Pathway overrepresentation analysis

In class, we learned about using the hypergeometric distribution to test for overrepresentation (or lack of independence) in the overlap of two groups. For example, we may wish to know if the set of differentially expressed genes overlaps more with the genes on a particular pathway than one might expect by chance alone.

Let's first familiarize ourselves with dhyper and phyper by reading the appropriate help page. Notice:

> `dhyper(x,m,n,k)` finds the probability of having *exactly* **x** white balls in your hand when drawing **k** from an urn with **m** white and **n** black. Note!! In R, the **n** is the *number of black balls* in the urn, not the total number of balls.

> `phyper(q,m,n,k)` finds the probability of getting *q or fewer* (i.e., less than or equal to q) white balls when **lower.tail=T** (default). For **lower.tail=F**, it gives the probability of getting *more than q* (i.e., strictly greater-than, not greater-or-equal) white balls.

Together, this means that if we want the probability of getting x or more white balls, we need to add them up. Let's make a function to do this:

```r
# x = num of white balls in hand (aka: num of DE genes in pathway)
# m = num of white balls in urn (aka: num of DE genes, total)
# Ntot = num of balls in urn (aka: num of genes tested)
# k = num of balls drawn (aka: num of genes on pathway)

overrep.pval <- function(x, m, Ntot, k){

    # First, let's get the number of black balls, which is what R wants
    n <- Ntot - m

    # Now let's get the probility of getting EXACTLY x white balls:
    p.exactly.x <- dhyper(x, m, n, k)
```

```
    # And then let's get the probability of getting MORE than x white:
    p.moreThan.x <- phyper(x, m, n, k, lower.tail = FALSE)

    # Add 'em up to get P(x or more; k, m, Ntot):
    pval <- p.exactly.x + p.moreThan.x

    return(pval)
}
```

Now let's try it out! This time, instead of using R's `$` notation to access a list element, we'll use the double-bracket notation `[[ ]]`. It's more typing, but you can use it in a loop (which you can't do with `$`), as we'll see later.

```
my.pathway <- kegg_pathways[["Complement and coagulation cascades"]]

# number of genes on the pathway
my.k <- length(my.pathway)

# total # of genes = balls in urn
my.Ntot <- length(DEgenes)

# total num of DE genes = white balls in urn
my.m <- sum(DEgenes)

my.x <- length(intersect( names(which(DEgenes)) , my.pathway ))

# run function
overrep.pval(my.x, my.m, my.Ntot, my.k)
```

```
## [1] 0.3944316
```

16. **Read the help page for `intersect`. What does the second to last line do?**

If we wanted to do the above for many pathways, it would be a pain to keep typing all that over and over. It would also be pretty inefficient, since we'd be needlessly re-computing `my.Ntot` and `my.m` each time, despite the fact that they never change. So let's make things easier on ourselves by defining a function:

```
# compute the global things that are the same regardless of pathway
N.tot.genes <- length(DEgenes) # total # of genes = balls in urn
m.DE.genes <- sum(DEgenes) # total # of DE genes = white balls in urn

# write a function to compute the pathway-specific stuff
# function takes in a pathway name
pathway.overrep.test <- function(pathwayName){
    # First, get the pathway. we've passed in the name as a variable, so use [[,]]:
    # no quotes! we're passing it as a variable.
    thisPathway <- kegg_pathways[[pathwayName]]

    # number of genes on the pathway
    k.pathway.length <- length(thisPathway)

    x.pathway.DE.overlap <- length(intersect( names(which(DEgenes)) , thisPathway ))

    pathway.p.val <- overrep.pval(x.pathway.DE.overlap,m.DE.genes,N.tot.genes,k.pathway.length)
```

```
    return(pathway.p.val)
}

# try it out!
pathway.overrep.test("Complement and coagulation cascades")
```

```
## [1] 0.3944316
```

Now we can use what we just wrote to loop over a nunch of pathways, and compute a p-val for each! Here's an example for the first five:

```
# initialize an empty vector to hold the results
first5pathway.pvals <- vector()

# names of the 1st 5 pathways
first5pathway.names <- names(kegg_pathways)[1:5]

for (pathwayName in first5pathway.names){
  first5pathway.pvals[pathwayName] <- pathway.overrep.test(pathwayName)
}

first5pathway.pvals
```

```
## Complement and coagulation cascades                Caffeine metabolism
##                           0.3944316                          1.0000000
##     Drug metabolism - other enzymes                 Metabolic pathways
##                           1.0000000                          1.0000000
##               Tryptophan metabolism
##                           1.0000000
```

17. **What does the line inside the for loop do? Explain all parts of it.**

18. **As a final exercise, we're going to test all the pathways!**

    a. **Modify the code above to get a p-value for all the pathways, and store them in a vector called `pathway_pvals`.**
    b. **How many pathways were significant with `p < 0.05`? Which pathways were they? Using R, spit out their p-values. Do the significant pathways surprise you (why or why not)?**
    c. **How many pathways did we test? Apply an appropriate multiple hypothesis correction.**
    d. **Repeat part (b) now that you've corrected. What would you conclude?**