Project 10 -- SEYI OGUNMODEDE

Instructor: Dr. Mark Daniel - Help with videos in figuring out how to write the functions. **Collaboration:** IK SU JEONG

Question 1

In [1]:
```
options(jupyter.rich_display = F)
```

In [2]:
```
options(repr.matrix.max.cols=50, repr.matrix.max.rows=200)
```

In [3]:
```
library(data.table)
```

In [4]:
```
users<- fread("/anvil/projects/tdm/data/okcupid/filtered/users.csv")

# fread, though will determine automatically what the dome meter are in the file
```

In [5]:
```
questions <- fread("/anvil/projects/tdm/data/okcupid/filtered/questions.csv")
# fread though will determine automatically what the dome meter are in the file
```

In [6]:
```
dim (users)
```

```
[1] 68371  2284
```

In [7]:
```
dim(questions)
```

```
[1] 2281   10
```

In [8]:
```
head(users)
```

```
  q41                 q42 q43 q45 q46         q48    q49      q50 q55 q56 q57
1 NA                  NA  NA  NA  NA          NA     Carefree NA  No  No  NA
2 NA                  NA  NA  NA  NA          NA     NA       NA  NA  NA  NA
3 Extremely important NA  NA  NA  Good        NA     Carefree NA  No  NA  NA
4 NA                  NA  NA  NA  NA          NA     NA       NA  NA  NA  NA
5 Not at all important NA  NA  NA  Interesting Weird  Intense  NA  NA  NA  NA
6 Not at all important NA  NA  NA  Good        Weird  NA       NA  NA  No  No
  q60          q61 q63 q65 q67 q68 q69 q70 q71 q73      q74      q76 q77
1 NA           NA  NA  NA  NA  NA  NA  No  No  NA       NA       NA  Sometimes
2 NA           NA  NA  NA  NA  NA  NA  NA  NA  NA       NA       NA  NA
3 Warm-hearted NA  NA  NA  NA  NA  NA  No  No  Big city Clothes  NA  Rarely
4 NA           NA  NA  NA  NA  NA  NA  NA  NA  NA       NA       NA  Sometimes
5 NA           NA  NA  NA  NA  NA  NA  No  NA  NA       NA       NA  Sometimes
6 NA           NA  NA  No  NA  NA  NA  No  No  NA       Food     NA  Sometimes
  q78 ... q85603 q85658 q85669 q85715 q85770 q85886 q85931 q85932 q85947 q86019
1 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
2 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
3 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
4 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
5 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
6 Yes ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
  q86210 q86215 q86283 q86325 q86364 q86397 q86462 q86699 q363047 CA
1 NA     NA     NA     NA     NA     NA     NA     NA     NA      0.7630798
2 NA     NA     NA     NA     NA     NA     NA     NA     NA             NA
3 NA     NA     NA     NA     NA     NA     NA     NA     NA      0.6613091
4 NA     NA     NA     NA     NA     NA     NA     NA     NA             NA
5 NA     NA     NA     NA     NA     NA     NA     NA     NA      0.8754240
6 NA     NA     NA     NA     NA     NA     NA     NA     NA      1.5153514
  gender_orientation race  gender2 gender2_num CA_items
1 Hetero_female      White Woman   0          4
2 Hetero_male        NA    Man     1          0
3 Hetero_female      NA    Woman   0          7
4 Hetero_female      White Woman   0          0
5 Bisexual_female    NA    Woman   0          3
6 Hetero_male        White Man     1          7
```

In [9]:
```
head(questions)
```

```
  X
1 q41
2 q42
3 q43
4 q44
5 q45
6 q46
  text
1 How important is religion/God in your life?
2 Is your duty to religion/God the most important thing in your life?
3 Do you believe in some kind of a spiritual, cosmic force outside the realm of most
modern religion?
4 Some religions are more correct than others.
5 Do you believe in ghosts?
6 Would you prefer good things happened, or interesting things?
  option_1            option_2           option_3
1 Extremely important Somewhat important Not very important
2 Yes                 No
3 Yes                 No
4 True                False
5 Yes                 No
6 Good                Interesting
  option_4              N     Type Order Keywords
1 Not at all important 54140 O          religion/superstition
2                       16024 O          religion/superstition
3                       6595  O          religion/superstition
4                       29109 O          religion/superstition
5                       21619 O          religion/superstition
6                       52112 N          preference
```

In [10]:
```
head(users$gender_orientation)
```

```
[1] "Hetero_female"   "Hetero_male"     "Hetero_female"   "Hetero_female"
[5] "Bisexual_female" "Hetero_male"
```

In [11]:
```
tail(users$gender_orientation)
```

```
[1] "Hetero_male"   "Hetero_male"   "Hetero_female" "Hetero_male"
[5] "Hetero_female" "Hetero_male"
```

In [12]:
```
head(users$race)
```

```
[1] "White" NA      NA      "White" NA      "White"
```

In [13]:
```
tail(users$race)
```

```
[1] "White" "White" NA      NA      "White" "Other"
```

In [14]:
```
head(questions$Type)
```

```
[1] "O" "O" "O" "O" "O" "N"
```

In [15]:
```
tail(questions$Type)
```

```
[1] "" "" "" "" "" ""
```

In [16]:
```
head(questions$text)
```

```
[1] "How important is religion/God in your life?"
[2] "Is your duty to religion/God the most important thing in your life?"
[3] "Do you believe in some kind of a spiritual, cosmic force outside the realm of mo
st modern religion?"
[4] "Some religions are more correct than others."
[5] "Do you believe in ghosts?"
[6] "Would you prefer good things happened, or interesting things?"
```

In [17]:
```
tail(questions$text)
```

```
[1] "Type of match"     "Max age of match"  "Looking for match"
[4] "Location of match" "Min age of match"  "Status of match"
```

We want to go ahead and load the datasets into data.frames named users and questions.

Take a look at both data.frames and identify what is a part of each of them.

user data frame has 68371 rows 2284 columns

questions data frame has 2281 rows 10 columns

What information is in each datatset, and how they are related?

For the the user, it is a data set about human being in respect to their gender and what they are interested in. for the question, it is a data set that has series of questions related to the user, in respect to texs, type, options, keywords as it applies.

Question 2

In [18]:
```
?grep
# grep helps to look for pattern inside text
# to see help function using grep
# to see other function the help using grep
# (in different ways to use grep)
```

grep package:base R Documentation

_ P_ a_ t_ t_ e_ r_ n _ M_ a_ t_ c_ h_ i_ n_ g _ a_ n_ d _ R_ e_ p_ l_ a_ c_ e_ m_ e_ n_ t

_ D_ e_ s_ c_ r_ i_ p_ t_ i_ o_ n:

'grep', 'grepl', 'regexpr', 'gregexpr', 'regexec' and 'gregexec' search for matches to argument 'pattern' within each element of a character vector: they differ in the format of and amount of detail in the results.

'sub' and 'gsub' perform replacement of the first and all matches respectively.

_ U_ s_ a_ g_ e:

```
grep(pattern, x, ignore.case = FALSE, perl = FALSE, value = FALSE,
     fixed = FALSE, useBytes = FALSE, invert = FALSE)

grepl(pattern, x, ignore.case = FALSE, perl = FALSE,
      fixed = FALSE, useBytes = FALSE)

sub(pattern, replacement, x, ignore.case = FALSE, perl = FALSE,
    fixed = FALSE, useBytes = FALSE)

gsub(pattern, replacement, x, ignore.case = FALSE, perl = FALSE,
     fixed = FALSE, useBytes = FALSE)

regexpr(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)

gregexpr(pattern, text, ignore.case = FALSE, perl = FALSE,
         fixed = FALSE, useBytes = FALSE)

regexec(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)

gregexec(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)
```

_ A_ r_ g_ u_ m_ e_ n_ t_ s:

```
 pattern: character string containing a regular expression (or
          character string for 'fixed = TRUE') to be matched in the
          given character vector.  Coerced by 'as.character' to a
          character string if possible.  If a character vector of
          length 2 or more is supplied, the first element is used with
          a warning.  Missing values are allowed except for 'regexpr',
          'gregexpr' and 'regexec'.

 x, text: a character vector where matches are sought, or an object
          which can be coerced by 'as.character' to a character vector.
          Long vectors are supported.

ignore.case: if 'FALSE', the pattern matching is _case sensitive_ and
          if 'TRUE', case is ignored during matching.

    perl: logical.  Should Perl-compatible regexps be used?
```

```
     value: if 'FALSE', a vector containing the ('integer') indices of
            the matches determined by 'grep' is returned, and if 'TRUE',
            a vector containing the matching elements themselves is
            returned.

     fixed: logical.  If 'TRUE', 'pattern' is a string to be matched as
            is.  Overrides all conflicting arguments.

 useBytes: logical.  If 'TRUE' the matching is done byte-by-byte rather
            than character-by-character.  See 'Details'.

    invert: logical.  If 'TRUE' return indices or values for elements
            that do _not_ match.

replacement: a replacement for matched pattern in 'sub' and 'gsub'.
            Coerced to character if possible.  For 'fixed = FALSE' this
            can include backreferences '"\1"' to '"\9"' to parenthesized
            subexpressions of 'pattern'.  For 'perl = TRUE' only, it can
            also contain '"\U"' or '"\L"' to convert the rest of the
            replacement to upper or lower case and '"\E"' to end case
            conversion.  If a character vector of length 2 or more is
            supplied, the first element is used with a warning.  If 'NA',
            all elements in the result corresponding to matches will be
            set to 'NA'.

_ D_ e_ t_ a_ i_ l_ s:

     Arguments which should be character strings or character vectors
     are coerced to character if possible.

     Each of these functions operates in one of three modes:

       1. 'fixed = TRUE': use exact matching.

       2. 'perl = TRUE': use Perl-style regular expressions.

       3. 'fixed = FALSE, perl = FALSE': use POSIX 1003.2 extended
          regular expressions (the default).

     See the help pages on regular expression for details of the
     different types of regular expressions.

     The two '*sub' functions differ only in that 'sub' replaces only
     the first occurrence of a 'pattern' whereas 'gsub' replaces all
     occurrences.  If 'replacement' contains backreferences which are
     not defined in 'pattern' the result is undefined (but most often
     the backreference is taken to be '""').

     For 'regexpr', 'gregexpr', 'regexec' and 'gregexec' it is an error
     for 'pattern' to be 'NA', otherwise 'NA' is permitted and gives an
     'NA' match.

     Both 'grep' and 'grepl' take missing values in 'x' as not matching
     a non-missing 'pattern'.

     The main effect of 'useBytes = TRUE' is to avoid errors/warnings
     about invalid inputs and spurious matches in multibyte locales,
     but for 'regexpr' it changes the interpretation of the output.  It
     inhibits the conversion of inputs with marked encodings, and is
     forced if any input is found which is marked as '"bytes"' (see
```

```
        'Encoding').

        Caseless matching does not make much sense for bytes in a
        multibyte locale, and you should expect it only to work for ASCII
        characters if 'useBytes = TRUE'.

        'regexpr' and 'gregexpr' with 'perl = TRUE' allow Python-style
        named captures, but not for _long vector_ inputs.

        Invalid inputs in the current locale are warned about up to 5
        times.

        Caseless matching with 'perl = TRUE' for non-ASCII characters
        depends on the PCRE library being compiled with 'Unicode property
        support', which PCRE2 is by default.

_ V_ a_ l_ u_ e:

        'grep(value = FALSE)' returns a vector of the indices of the
        elements of 'x' that yielded a match (or not, for 'invert =
        TRUE').  This will be an integer vector unless the input is a
        _long vector_, when it will be a double vector.

        'grep(value = TRUE)' returns a character vector containing the
        selected elements of 'x' (after coercion, preserving names but no
        other attributes).

        'grepl' returns a logical vector (match or not for each element of
        'x').

        'sub' and 'gsub' return a character vector of the same length and
        with the same attributes as 'x' (after possible coercion to
        character).  Elements of character vectors 'x' which are not
        substituted will be returned unchanged (including any declared
        encoding).  If 'useBytes = FALSE' a non-ASCII substituted result
        will often be in UTF-8 with a marked encoding (e.g., if there is a
        UTF-8 input, and in a multibyte locale unless 'fixed = TRUE').
        Such strings can be re-encoded by 'enc2native'.

        'regexpr' returns an integer vector of the same length as 'text'
        giving the starting position of the first match or -1 if there is
        none, with attribute '"match.length"', an integer vector giving
        the length of the matched text (or -1 for no match).  The match
        positions and lengths are in characters unless 'useBytes = TRUE'
        is used, when they are in bytes (as they are for ASCII-only
        matching: in either case an attribute 'useBytes' with value 'TRUE'
        is set on the result).  If named capture is used there are further
        attributes '"capture.start"', '"capture.length"' and
        '"capture.names"'.

        'gregexpr' returns a list of the same length as 'text' each
        element of which is of the same form as the return value for
        'regexpr', except that the starting positions of every (disjoint)
        match are given.

        'regexec' returns a list of the same length as 'text' each element
        of which is either -1 if there is no match, or a sequence of
        integers with the starting positions of the match and all
        substrings corresponding to parenthesized subexpressions of
        'pattern', with attribute '"match.length"' a vector giving the
```

```
     lengths of the matches (or -1 for no match).  The interpretation
     of positions and length and the attributes follows 'regexpr'.

     'gregexec' returns the same as 'regexec', except that to
     accommodate multiple matches per element of 'text', the integer
     sequences for each match are made into columns of a matrix, with
     one matrix per element of 'text' with matches.

     Where matching failed because of resource limits (especially for
     'perl = TRUE') this is regarded as a non-match, usually with a
     warning.

_ W_ a_ r_ n_ i_ n_ g:

     The POSIX 1003.2 mode of 'gsub' and 'gregexpr' does not work
     correctly with repeated word-boundaries (e.g., 'pattern = "\b"').
     Use 'perl = TRUE' for such matches (but that may not work as
     expected with non-ASCII inputs, as the meaning of 'word' is
     system-dependent).

_ P_ e_ r_ f_ o_ r_ m_ a_ n_ c_ e _ c_ o_ n_ s_ i_ d_ e_ r_ a_ t_ i_ o_ n_ s:

     If you are doing a lot of regular expression matching, including
     on very long strings, you will want to consider the options used.
     Generally 'perl = TRUE' will be faster than the default regular
     expression engine, and 'fixed = TRUE' faster still (especially
     when each pattern is matched only a few times).

     If you are working in a single-byte locale and have marked UTF-8
     strings that are representable in that locale, convert them first
     as just one UTF-8 string will force all the matching to be done in
     Unicode, which attracts a penalty of around 3x for the default
     POSIX 1003.2 mode.

     If you can make use of 'useBytes = TRUE', the strings will not be
     checked before matching, and the actual matching will be faster.
     Often byte-based matching suffices in a UTF-8 locale since byte
     patterns of one character never match part of another.  Character
     ranges may produce unexpected results.

     PCRE-based matching by default used to put additional effort into
     'studying' the compiled pattern when 'x'/'text' has length 10 or
     more.  That study may use the PCRE JIT compiler on platforms where
     it is available (see 'pcre_config'). As from PCRE2 (PCRE version
     >= 10.00 as reported by 'extSoftVersion'), there is no study
     phase, but the patterns are optimized automatically when possible,
     and PCRE JIT is used when enabled.  The details are controlled by
     'options' 'PCRE_study' and 'PCRE_use_JIT'.  (Some timing
     comparisons can be seen by running file 'tests/PCRE.R' in the R
     sources (and perhaps installed).)  People working with PCRE and
     very long strings can adjust the maximum size of the JIT stack by
     setting environment variable 'R_PCRE_JIT_STACK_MAXSIZE' before JIT
     is used to a value between '1' and '1000' in MB: the default is
     '64'. When JIT is not used with PCRE version < 10.30 (that is with
     PCRE1 and old versions of PCRE2), it might also be wise to set the
     option 'PCRE_limit_recursion'.

_ N_ o_ t_ e:

     Aspects will be platform-dependent as well as locale-dependent:
```

```
     for example the implementation of character classes (except
     '[:digit:]' and '[:xdigit:]').  One can expect results to be
     consistent for ASCII inputs and when working in UTF-8 mode (when
     most platforms will use Unicode character tables, although those
     are updated frequently and subject to some degree of
     interpretation - is a circled capital letter alphabetic or a
     symbol?).  However, results in 8-bit encodings can differ
     considerably between platforms, modes and from the UTF-8 versions.

_ S_ o_ u_ r_ c_ e:

     The C code for POSIX-style regular expression matching has changed
     over the years.  As from R 2.10.0 (Oct 2009) the TRE library of
     Ville Laurikari (<https://github.com/laurikari/tre>) is used.  The
     POSIX standard does give some room for interpretation, especially
     in the handling of invalid regular expressions and the collation
     of character ranges, so the results will have changed slightly
     over the years.

     For Perl-style matching PCRE2 or PCRE (<https://www.pcre.org>) is
     used: again the results may depend (slightly) on the version of
     PCRE in use.

_ R_ e_ f_ e_ r_ e_ n_ c_ e_ s:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_.  Wadsworth & Brooks/Cole ('grep')

_ S_ e_ e _ A_ l_ s_ o:

     regular expression (aka 'regexp') for the details of the pattern
     specification.

     'regmatches' for extracting matched substrings based on the
     results of 'regexpr', 'gregexpr' and 'regexec'.

     'glob2rx' to turn wildcard matches into regular expressions.

     'agrep' for approximate matching.

     'charmatch', 'pmatch' for partial matching, 'match' for matching
     to whole strings, 'startsWith' for matching of initial parts of
     strings.

     'tolower', 'toupper' and 'chartr' for character translations.

     'apropos' uses regexps and has more examples.

     'grepRaw' for matching raw vectors.

     Options 'PCRE_limit_recursion', 'PCRE_study' and 'PCRE_use_JIT'.

     'extSoftVersion' for the versions of regex and PCRE libraries in
     use, 'pcre_config' for more details for PCRE.

_ E_ x_ a_ m_ p_ l_ e_ s:

     grep("[a-z]", letters)

     txt <- c("arm","foot","lefroo", "bafoobar")
```

```
if(length(i <- grep("foo", txt)))
   cat("'foo' appears at least once in\n\t", txt, "\n")
i # 2 and 4
txt[i]

## Double all 'a' or 'b's;  "\" must be escaped, i.e., 'doubled'
gsub("([ab])", "\\1_\\1_", "abc and ABC")

txt <- c("The", "licenses", "for", "most", "software", "are",
  "designed", "to", "take", "away", "your", "freedom",
  "to", "share", "and", "change", "it.",
  "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
  "is", "intended", "to", "guarantee", "your", "freedom", "to",
  "share", "and", "change", "free", "software", "--",
  "to", "make", "sure", "the", "software", "is",
  "free", "for", "all", "its", "users")
( i <- grep("[gu]", txt) ) # indices
stopifnot( txt[i] == grep("[gu]", txt, value = TRUE) )

## Note that for some implementations character ranges are
## locale-dependent (but not currently).  Then [b-e] in locales such as
## en_US may include B as the collation order is aAbBcCdDe ...
(ot <- sub("[b-e]",".", txt))
txt[ot != gsub("[b-e]",".", txt)]#- gsub does "global" substitution
## In caseless matching, ranges include both cases:
a <- grep("[b-e]", txt, value = TRUE)
b <- grep("[b-e]", txt, ignore.case = TRUE, value = TRUE)
setdiff(b, a)

txt[gsub("g","#", txt) !=
    gsub("g","#", txt, ignore.case = TRUE)] # the "G" words

regexpr("en", txt)

gregexpr("e", txt)

## Using grepl() for filtering
## Find functions with argument names matching "warn":
findArgs <- function(env, pattern) {
  nms <- ls(envir = as.environment(env))
  nms <- nms[is.na(match(nms, c("F","T")))] # <-- work around "checking hack"
  aa <- sapply(nms, function(.) { o <- get(.)
              if(is.function(o)) names(formals(o)) })
  iw <- sapply(aa, function(a) any(grepl(pattern, a, ignore.case=TRUE)))
  aa[iw]
}
findArgs("package:base", "warn")

## trim trailing white space
str <- "Now is the time      "
sub(" +$", "", str)  ## spaces only
## what is considered 'white space' depends on the locale.
sub("[[:space:]]+$", "", str) ## white space, POSIX-style
## what PCRE considered white space changed in version 8.34: see ?regex
sub("\\s+$", "", str, perl = TRUE) ## PCRE-style white space

## capitalizing
txt <- "a test of capitalizing"
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2", txt, perl=TRUE)
gsub("\\b(\\w)",    "\\U\\1",       txt, perl=TRUE)
```

```
txt2 <- "useRs may fly into JFK or laGuardia"
gsub("(\\w)(\\w*)(\\w)", "\\U\\1\\E\\2\\U\\3", txt2, perl=TRUE)
 sub("(\\w)(\\w*)(\\w)", "\\U\\1\\E\\2\\U\\3", txt2, perl=TRUE)

## named capture
notables <- c("  Ben Franklin and Jefferson Davis",
              "\tMillard Fillmore")
# name groups 'first' and 'last'
name.rex <- "(?<first>[[:upper:]][[:lower:]]+) (?<last>[[:upper:]][[:lower:]]+)"
(parsed <- regexpr(name.rex, notables, perl = TRUE))
gregexpr(name.rex, notables, perl = TRUE)[[2]]
parse.one <- function(res, result) {
  m <- do.call(rbind, lapply(seq_along(res), function(i) {
    if(result[i] == -1) return("")
    st <- attr(result, "capture.start")[i, ]
    substring(res[i], st, st + attr(result, "capture.length")[i, ] - 1)
  }))
  colnames(m) <- attr(result, "capture.names")
  m
}
parse.one(notables, parsed)

## Decompose a URL into its components.
## Example by LT (http://www.cs.uiowa.edu/~luke/R/regexp.html).
x <- "http://stat.umn.edu:80/xyz"
m <- regexec("^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)", x)
m
regmatches(x, m)
## Element 3 is the protocol, 4 is the host, 6 is the port, and 7
## is the path.  We can use this to make a function for extracting the
## parts of a URL:
URL_parts <- function(x) {
    m <- regexec("^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)", x)
    parts <- do.call(rbind,
                     lapply(regmatches(x, m), `[`, c(3L, 4L, 6L, 7L)))
    colnames(parts) <- c("protocol","host","port","path")
    parts
}
URL_parts(x)

## gregexec() may match multiple times within a single string.
pattern <- "([[:alpha:]]+)([[:digit:]]+)"
s <- "Test: A1 BC23 DEF456"
m <- gregexec(pattern, s)
m
regmatches(s, m)

## Before gregexec() was implemented, one could emulate it by running
## regexec() on the regmatches obtained via gregexpr().  E.g.:
lapply(regmatches(s, gregexpr(pattern, s)),
       function(e) regmatches(e, regexec(pattern, e)))
```

In [19]:
```
dim(questions)
# it has 10 columns
```

```
[1] 2281   10
```

In [20]:
```
head(questions)
# to see the word goooogle, it may be possible with text
```

```
  X
1 q41
2 q42
3 q43
4 q44
5 q45
6 q46
  text
1 How important is religion/God in your life?
2 Is your duty to religion/God the most important thing in your life?
3 Do you believe in some kind of a spiritual, cosmic force outside the realm of most
modern religion?
4 Some religions are more correct than others.
5 Do you believe in ghosts?
6 Would you prefer good things happened, or interesting things?
  option_1            option_2           option_3
1 Extremely important Somewhat important Not very important
2 Yes                 No
3 Yes                 No
4 True                False
5 Yes                 No
6 Good                Interesting
  option_4              N     Type Order Keywords
1 Not at all important 54140 O          religion/superstition
2                       16024 O          religion/superstition
3                       6595  O          religion/superstition
4                       29109 O          religion/superstition
5                       21619 O          religion/superstition
6                       52112 N          preference
```

In [21]:
```
grep("Google",questions$text)
# it does find it one time on row 2172
```

```
[1] 2172
```

In [22]:
```
questions[2172, ]
# to detect what is in the column
```

```
  X       text                                          option_1
1 q170849 Do you Google someone before a first date? Yes. Knowledge is power!
  option_2                  option_3 option_4 N     Type Order
1 No. Why spoil the mystery?                  39621 O
  Keywords
1 descriptive; technology
```

In [23]:
```
questions$text[grepl("Google",questions$text)]
# another way is using logical grep (grepl)
# a neat thing about grepl is use the location
# at the begining as a position within the vector of text within the text column
```

```
[1] "Do you Google someone before a first date?"
```

What do you notice if you just use the function grep() and create a new variable google and then print that variable? it helps to locate the number of occurence and the row it appears.

Now that you know the row number, how can you take a look at the information there?

X text option_1
1 q170849 Do you Google someone before a first date? Yes. Knowledge is power! option_2
option_3 option_4 N Type Order 1 No. Why spoil the mystery? 39621 O
Keywords
1 descriptive; technology

using grepl

"Do you Google someone before a first date?"

Question 3

In [24]:
```
dim(users)
# 68371rows  2284columns
# roughly one colmn per question
```

```
[1] 68371  2284
```

In [25]:
```
head(users)
# showing one column per question
```

```
  q41                  q42 q43 q45 q46         q48    q49       q50 q55 q56 q57
1 NA                   NA  NA  NA  NA          NA     Carefree  NA  No  No  NA
2 NA                   NA  NA  NA  NA          NA     NA        NA  NA  NA  NA
3 Extremely important  NA  NA  NA  Good        NA     Carefree  NA  No  NA  NA
4 NA                   NA  NA  NA  NA          NA     NA        NA  NA  NA  NA
5 Not at all important NA  NA  NA  Interesting Weird  Intense   NA  NA  NA  NA
6 Not at all important NA  NA  NA  Good        Weird  NA        NA  NA  No  No
  q60          q61 q63 q65 q67 q68 q69 q70 q71 q73      q74      q76 q77
1 NA           NA  NA  NA  NA  NA  NA  No  No  NA       NA       NA  Sometimes
2 NA           NA  NA  NA  NA  NA  NA  NA  NA  NA       NA       NA  NA
3 Warm-hearted NA  NA  NA  NA  NA  NA  No  No  Big city Clothes  NA  Rarely
4 NA           NA  NA  NA  NA  NA  NA  NA  NA  NA       NA       NA  Sometimes
5 NA           NA  NA  NA  NA  NA  NA  No  NA  NA       NA       NA  Sometimes
6 NA           NA  NA  No  NA  NA  NA  No  No  NA       Food     NA  Sometimes
  q78 ... q85603 q85658 q85669 q85715 q85770 q85886 q85931 q85932 q85947 q86019
1 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
2 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
3 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
4 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
5 NA  ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
6 Yes ... NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
  q86210 q86215 q86283 q86325 q86364 q86397 q86462 q86699 q363047 CA
1 NA     NA     NA     NA     NA     NA     NA     NA     NA      0.7630798
2 NA     NA     NA     NA     NA     NA     NA     NA     NA             NA
3 NA     NA     NA     NA     NA     NA     NA     NA     NA      0.6613091
4 NA     NA     NA     NA     NA     NA     NA     NA     NA             NA
5 NA     NA     NA     NA     NA     NA     NA     NA     NA      0.8754240
6 NA     NA     NA     NA     NA     NA     NA     NA     NA      1.5153514
  gender_orientation race  gender2 gender2_num CA_items
1 Hetero_female      White Woman   0          4
2 Hetero_male        NA    Man     1          0
3 Hetero_female      NA    Woman   0          7
4 Hetero_female      White Woman   0          0
5 Bisexual_female    NA    Woman   0          3
6 Hetero_male        White Man     1          7
```

In [26]:
```
which(names(users) == "q170849")
# to look for which one is TRUE
#out of the users  data frame, colmn 1193 is TRUE
```

```
[1] 1193
```

In [27]:
```
users[ ,1193]
# to detect what is in the row
```

	q170849
1	No. Why spoil the mystery?
2	NA
3	No. Why spoil the mystery?
4	NA
5	NA
6	Yes. Knowledge is power!
7	Yes. Knowledge is power!
8	NA
9	NA
10	NA
11	No. Why spoil the mystery?
12	NA
13	Yes. Knowledge is power!
14	NA
15	No. Why spoil the mystery?
16	NA
17	NA
18	NA
19	No. Why spoil the mystery?
20	NA
21	NA
22	No. Why spoil the mystery?
23	Yes. Knowledge is power!
24	Yes. Knowledge is power!
25	NA
26	No. Why spoil the mystery?
27	NA
28	NA
29	NA
30	No. Why spoil the mystery?
31	Yes. Knowledge is power!
32	NA
33	No. Why spoil the mystery?
34	Yes. Knowledge is power!
35	NA
36	Yes. Knowledge is power!
37	NA
38	Yes. Knowledge is power!
39	No. Why spoil the mystery?
40	No. Why spoil the mystery?
41	Yes. Knowledge is power!
42	NA
43	NA
44	NA
45	Yes. Knowledge is power!
46	NA
47	NA
48	NA
49	No. Why spoil the mystery?
50	NA
51	No. Why spoil the mystery?
52	NA
53	NA
54	NA
55	NA
56	No. Why spoil the mystery?
57	NA
58	Yes. Knowledge is power!
59	NA

```
60    NA
61    Yes. Knowledge is power!
62    NA
63    NA
64    NA
65    NA
66    Yes. Knowledge is power!
67    NA
68    NA
69    NA
70    NA
71    No. Why spoil the mystery?
72    NA
73    No. Why spoil the mystery?
74    NA
75    NA
76    NA
77    NA
78    No. Why spoil the mystery?
79    NA
80    NA
81    No. Why spoil the mystery?
82    Yes. Knowledge is power!
83    NA
84    No. Why spoil the mystery?
85    NA
86    No. Why spoil the mystery?
87    NA
88    NA
89    NA
90    NA
91    NA
92    NA
93    Yes. Knowledge is power!
94    Yes. Knowledge is power!
95    NA
96    NA
97    Yes. Knowledge is power!
98    No. Why spoil the mystery?
99    NA
100   NA
...   ...
68272 NA
68273 NA
68274 NA
68275 NA
68276 NA
68277 NA
68278 NA
68279 NA
68280 NA
68281 No. Why spoil the mystery?
68282 NA
68283 NA
68284 No. Why spoil the mystery?
68285 NA
68286 NA
68287 NA
68288 NA
68289 No. Why spoil the mystery?
```

```
68290 NA
68291 NA
68292 NA
68293 NA
68294 NA
68295 NA
68296 NA
68297 No. Why spoil the mystery?
68298 NA
68299 NA
68300 NA
68301 NA
68302 NA
68303 NA
68304 NA
68305 NA
68306 Yes. Knowledge is power!
68307 Yes. Knowledge is power!
68308 NA
68309 NA
68310 NA
68311 NA
68312 NA
68313 NA
68314 NA
68315 NA
68316 No. Why spoil the mystery?
68317 NA
68318 No. Why spoil the mystery?
68319 NA
68320 NA
68321 NA
68322 NA
68323 NA
68324 NA
68325 NA
68326 NA
68327 No. Why spoil the mystery?
68328 NA
68329 NA
68330 NA
68331 NA
68332 NA
68333 NA
68334 NA
68335 NA
68336 NA
68337 NA
68338 NA
68339 NA
68340 NA
68341 NA
68342 NA
68343 NA
68344 NA
68345 NA
68346 NA
68347 NA
68348 NA
68349 NA
```

```
68350 NA
68351 NA
68352 NA
68353 NA
68354 NA
68355 NA
68356 NA
68357 NA
68358 NA
68359 NA
68360 NA
68361 NA
68362 NA
68363 NA
68364 NA
68365 NA
68366 NA
68367 NA
68368 NA
68369 NA
68370 NA
68371 NA
```

In [28]:
```
table(users[[1193]], useNA = "always")
# another way to look at it
# because the outcome contain NA's,
# then use table and use NA to know how many NA are there
```

```
No. Why spoil the mystery?   Yes. Knowledge is power!
                     25804                       13817
                      <NA>
                     28750
```

In [29]:
```
prop.table(table(users[[1193]], useNA = "always"))
# this shows the proportion
```

```
No. Why spoil the mystery?   Yes. Knowledge is power!
                 0.3774115                   0.2020886
                      <NA>
                 0.4204999
```

In [30]:
```
(prop.table(table(users[[1193]], useNA = "always")))*100
# to know the actual percentage by multiplying by 100
```

```
No. Why spoil the mystery?   Yes. Knowledge is power!
                  37.74115                    20.20886
                      <NA>
                  42.04999
```

In [31]:
```
users[2172 ,1193]
# for verification
```

```
  q170849
1 No. Why spoil the mystery?
```

Using the row from our previous question, which variable does this correspond with in the data.frame users? colmn 1193 Row 2172 in questions corresponds to column named q170849 in users

Knowing that the two possible answers are "No. Why spoil the mystery?" and "Yes, Knowledge is power!" What percentage of users do NOT google someone before the first date? 37.74%

Question 4

In [ ]:
```
tapply(users[[1193]], gender, table)
# we want to break things up using the table
# we are yet to look at the gender
# colomn to consider that is for gender is (gender 2 in column 2282)
```

In [33]:
```
head(users[[2282]])
# these are the genders there.
# the column named gender2 is 2282
```

```
[1] "Woman" "Man"   "Woman" "Woman" "Woman" "Man"
```

In [34]:
```
tapply(users[[1193]], users[[2282]], table)
# we are going to take the data from how people responded to the Google questions
# break it up according to the gender
# within all the males responses, we take the data and make a table
# within all the females responses, we take the data and make a table
```

```
$Man

No. Why spoil the mystery?   Yes. Knowledge is power!
                     17294                       7549

$Woman

No. Why spoil the mystery?   Yes. Knowledge is power!
                      7987                       5660
```

In [35]:
```
tapply(users[[1193]], users[[2282]], table,useNA = "always")
# put useNA = "always" to know the number of
# people that did not respond in each gender
```

```
$Man

No. Why spoil the mystery?   Yes. Knowledge is power!
                     17294                       7549
                      <NA>
                     15372

$Woman

No. Why spoil the mystery?   Yes. Knowledge is power!
                      7987                       5660
                      <NA>
                     12305
```

In [36]:
```
tapply(users[[1193]], users[[2282]], function(x)
{prop.table(table(x,useNA = "always"))})
# another way to calculate the percentage using prop.table
```

```
$Man
x
No. Why spoil the mystery?   Yes. Knowledge is power!
                 0.4300385                   0.1877160
                      <NA>
                 0.3822454

$Woman
x
No. Why spoil the mystery?   Yes. Knowledge is power!
                 0.3077605                   0.2180949
                      <NA>
                 0.4741446
```

In [37]:
```
tapply(users[[1193]], users[[2282]], function(x)
{(prop.table(table(x,useNA = "always")))*100})
```

```
$Man
x
No. Why spoil the mystery?   Yes. Knowledge is power!
                  43.00385                    18.77160
                      <NA>
                  38.22454

$Woman
x
No. Why spoil the mystery?   Yes. Knowledge is power!
                  30.77605                    21.80949
                      <NA>
                  47.41446
```

In [38]:
```
sapply(tapply(users[[1193]], users[[2282]],
              table,useNA = "always"),prop.table)
# this apply prop.table to each
# this take data from men and make proportion to women
```

```
                         Man       Woman
No. Why spoil the mystery? 0.4300385 0.3077605
Yes. Knowledge is power!   0.1877160 0.2180949
<NA>                       0.3822454 0.4741446
```

In [ ]:
```
# sapply works like this:    sapply(mydata, myfunction)
# sapply takes some data and a function
# that you want to run on each piece of the data
```

Using the ability to create a function AND tapply find the percentages of Female vs Male (Man vs Woman, as categorized in the users data.frame) who DO google someone before their date.

Question 5

In [39]:
```
head(users$gender_orientation)
```

```
[1] "Hetero_female"   "Hetero_male"     "Hetero_female"   "Hetero_female"
[5] "Bisexual_female" "Hetero_male"
```

In [40]:
```
head(users$gender_orientation, n=20)
```

```
 [1] "Hetero_female"   "Hetero_male"     "Hetero_female"   "Hetero_female"
 [5] "Bisexual_female" "Hetero_male"     NA                "Hetero_male"
 [9] "Hetero_female"   "Hetero_male"     "Hetero_female"   "Hetero_male"
[13] "Bisexual_female" "Hetero_female"   "Hetero_male"     "Bisexual_female"
[17] NA                "Hetero_female"   "Hetero_male"     "Hetero_female"
```

In [41]:
```
gsub("_", "", head(users$gender_orientation, n=20))
# take the underscore out and everything after underscore
# to do that, i do a global substitution of the
# underscore and put in nothing in its place
# it will just take out the underscore itself
# but it wont get things after the underscore
# lots of way to solve the problems
```

```
 [1] "Heterofemale"   "Heteromale"     "Heterofemale"   "Heterofemale"
 [5] "Bisexualfemale" "Heteromale"     NA               "Heteromale"
 [9] "Heterofemale"   "Heteromale"     "Heterofemale"   "Heteromale"
[13] "Bisexualfemale" "Heterofemale"   "Heteromale"     "Bisexualfemale"
[17] NA               "Heterofemale"   "Heteromale"     "Heterofemale"
```

In [42]:
```
gsub("_[A-Za-z]*", "", head(users$gender_orientation, n=20))
# i will take any letter that comes after that
# [A-Za-z] means take any letter capitals or smallletters that comes after underscore
# * means  take as many of those you find (by putting *)
#  "" means substituting them out to make them into nothing
```

```
 [1] "Hetero"   "Hetero"   "Hetero"   "Hetero"   "Bisexual" "Hetero"
 [7] NA         "Hetero"   "Hetero"   "Hetero"   "Hetero"   "Hetero"
[13] "Bisexual" "Hetero"   "Hetero"   "Bisexual" NA         "Hetero"
[19] "Hetero"   "Hetero"
```

In [43]:
```
gsub("_[A-Za-z]*", "burger", head(users$gender_orientation, n=20))
# the thing you put in here ("") is what get sustituted
# if you put burger in it (e.g "burger")
```

```
 [1] "Heteroburger"   "Heteroburger"   "Heteroburger"   "Heteroburger"
 [5] "Bisexualburger" "Heteroburger"   NA               "Heteroburger"
 [9] "Heteroburger"   "Heteroburger"   "Heteroburger"   "Heteroburger"
[13] "Bisexualburger" "Heteroburger"   "Heteroburger"   "Bisexualburger"
[17] NA               "Heteroburger"   "Heteroburger"   "Heteroburger"
```

In [44]:
```
gsub("_[A-Za-z]*", "", head(users$gender_orientation, n=20))
# we want to remove everything after the underscore
```

```
 [1] "Hetero"   "Hetero"   "Hetero"   "Hetero"   "Bisexual" "Hetero"
 [7] NA         "Hetero"   "Hetero"   "Hetero"   "Hetero"   "Hetero"
[13] "Bisexual" "Hetero"   "Hetero"   "Bisexual" NA         "Hetero"
[19] "Hetero"   "Hetero"
```

In [45]:
```
table(gsub("_[A-Za-z]*", "", head(users$gender_orientation, n=20)))
# then make table of the result
```

```
Bisexual   Hetero
       3       15
```

In [46]:
```
table(gsub("_[A-Za-z]*", "", head(users$gender_orientation, n=100)))
# table for the first 100 entries
```

```
Bisexual      Gay   Hetero
      12        4       78
```

In [47]:
```
table(gsub("_[A-Za-z]*", "", users$gender_orientation), useNA="always")
# since i have gotten for first 100 entries
# for all together i will take the head off, and the range(n=100)
# lastly you could check, is there any NA's in there (useNA="always")
```

```
Bisexual      Gay   Hetero     <NA> 
    4743     3392    57747     2489 
```

Using the ability to create a function AND use sapply() write a function that takes the string and removes everything after/including the _ from the gender_orientation column in the users data.frame.

Pledge

By submitting this work I hereby pledge that this is my own, personal work. I've acknowledged in the designated place at the top of this file all sources that I used to complete said work, including but not limited to: online resources, books, and electronic communications. I've noted all collaboration with fellow students and/or TA's. I did not copy or plagiarize another's work.

> As a Boilermaker pursuing academic excellence, I pledge to be honest and true in all that I do. Accountable together – We are Purdue.