Day 4 Lecture

Cecilia Y. Sui

01/13/2022

Day 4 Outline:

1. Types of Statistical Data (numerical, categorical, ordinal)
2. Probability distributions: rnorm / dnorm / pnorm / qnorm
3. Descriptive Statistics

1. Types of Statistical Data in R

The most common variables used in data analysis can be classified as one of three types of variables: numerical, categorical/nominal and ordinal data.

Understanding the differences in these types of variables is critical, since the variable type will determine which statistical analysis will be valid for that data. In addition, the way we summarize data with statistics and plots will be determined by the variable type.

1.1 Numerical Data

Numerical data are usually from interval or ratio variables that are measured or counted values, such as age, height, weight, number of votes. The interval between numbers is known to be equal.

Numerical data can be either discrete or continuous. Counted data are usually discrete, while measured data are usually continuous.

1.2 Categorical Data

Categorical or nominal variables are data whose levels are labels or descriptions, and which cannot be ordered. Examples of categorical variables are sex, school, and yes/no questions.They are also called "nominal categorical" or "qualitative" variables, and the levels of a variable are sometimes called "classes" or "groups".

The levels of categorical variables cannot be ordered. For the variable sex, it makes no sense to try to put the levels "female", "male", and "other" in any numerical order. If levels are numbered for convenience, the numbers are arbitrary, and the variable can't be treated as a numeric variable. Moreover, taking the average of such numbers is often not meaningful.

1.3 Ordinal Data

Ordinal variables can be ordered, or ranked in logical order, but the interval between levels of the variables are not necessarily known. Subjective measurements are often ordinal variables. One example would be having people rank four items by preference in order from one to four. A different example would be having people assess several items based on a Likert ranking scale: “On a scale of one to five, do you agree or disagree with this statement?” A third example is level of education for adults, considering for example “less than high school”, “high school”, “associate’s degree,” etc.

Critically, in each case we can order the responses, but we cannot know if the interval between the levels is equal. For example, the distance between your favorite salad dressing and your second favorite salad dressing may be small, where there may be a large gap between your second and third choices.

We can logically assign numbers to levels of an ordinal variable, and can treat them in order, but should NOT treat them as numeric: “strongly agree” and “neutral” may not average out to an “agree.”

For the purpose of the bootcamp,we will consider such Likert item data to be ordinal data under most circumstances. Ordinal data is sometimes called “ordered categorical”.

1.4 Types of Statistical Data in R

Unfortunately, R does not use the terms numerical, categorical, and ordinal for types of variables.

Numerical variables can be coded as variables with numeric or integer classes. The default class to store numbers in R is numeric. An L used with values to tell R to store the data as an integer class.

```
Count <- c(1, 2, 3, 4, 5)
class(Count)
```

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

```
Count.int <- c(1L, 2L, 3L, 4L, 5L)
class(Count.int)
```

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

R has two names for its floating-point vector: “double” and “numeric”.

“double” is the name of the type, and “numeric” is the name of the class.

In R, categorical or nominal variables can be coded as variables with factor or character classes. The **factor()** command is used to create and modify factors in R.

For example:

```
Colors <- c("Red", "Green", "Blue")
class(Colors)
```

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

```
Colors.f <- factor(c("Red", "Green", "Blue"))
class(Colors.f)
```

```
## [1] "factor"
```

Factors are stored as integers in R, and have labels associated with these unique integers. While factors look and often behave like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a **pre-defined set values**, which are the **levels**. By default, R always sorts levels in **alphabetical** order. For example:

```r
sex <- factor(c("male", "female", "female", "male"))
levels(sex)
```

```
## [1] "female" "male"
```

R will assign 1 to the level "female" and 2 to the level "male" (because f comes before m alphabetically, even though the first element in this vector is "male"). You can check this by using the function **levels()**, and check the number of levels using **nlevels()**:

```r
levels(sex)
```

```
## [1] "female" "male"
```

```r
nlevels(sex)
```

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

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "medium", "high") or it is required by particular type of analysis. Additionally, specifying the order of the levels allows us to compare levels:

```r
TFR.f <- factor(c("low", "high", "medium", "high", "low", "medium", "high"))
levels(TFR.f)
```

```
## [1] "high"   "low"    "medium"
```

```r
TFR.f1 <- factor(TFR.f, levels=c("low", "medium", "high"))
levels(TFR.f1)
```

```
## [1] "low"    "medium" "high"
```

However, once you have ordered your levels, it would be great if you can find the lowest or highest level.

```r
min(TFR.f1) # does not work
```

You can address that error by adding another argument in your **factor()** function:

```r
TFR.f1 <- factor(TFR.f, levels=c("low", "medium", "high"), ordered = TRUE)
min(TFR.f1)
```

```
## [1] low
## Levels: low < medium < high
```

```r
max(TFR.f1)
```

```
## [1] high
## Levels: low < medium < high
```

In R's memory, these factors are represented by numbers (1, 2, 3). They are better than using simple integer labels because factors are self describing: "low", "medium", and "high"" is more descriptive than 1, 2, 3. Which is low? You wouldn't be able to tell with just integer data. Factors have this information built in. It is particularly helpful when there are many levels.

We can code ordinal data as either numeric or factor variables, depending on how we will be summarizing, plotting, and analyzing it.

When we load a dataset into R using the read.csv function, R does not always store them with the desired classes. For example, in our school location dataset, R stores the column "STATE" as characters. It would make more sense to treat them as factors. We can convert them to factors like this:

```
school_loc <- read.csv("school_loc.csv")
# summary(school_loc)
str(school_loc)
```

```
## 'data.frame':    7012 obs. of  26 variables:
##  $ X         : num  -86.6 -86.8 -86.2 -86.6 -86.3 ...
##  $ Y         : num  34.8 33.5 32.4 34.7 32.4 ...
##  $ OBJECTID  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ UNITID    : int  100654 100663 100690 100706 100724 100733 100751 100760 100812 100830 ...
##  $ NAME      : chr  "Alabama A & M University" "University of Alabama at Birmingham" "Amridge Univers
##  $ STREET    : chr  "4900 Meridian Street" "Administration Bldg Suite 1070" "1200 Taylor Rd" "301 Spa
##  $ CITY      : chr  "Normal" "Birmingham" "Montgomery" "Huntsville" ...
##  $ STATE     : chr  "AL" "AL" "AL" "AL" ...
##  $ ZIP       : chr  "35762" "35294-0110" "36117-3553" "35899" ...
##  $ STFIP     : chr  "01" "01" "01" "01" ...
##  $ CNTY      : chr  "01089" "01073" "01101" "01089" ...
##  $ NMCNTY    : chr  "Madison County" "Jefferson County" "Montgomery County" "Madison County" ...
##  $ LOCALE    : chr  "12" "12" "12" "12" ...
##  $ LAT       : num  34.8 33.5 32.4 34.7 32.4 ...
##  $ LON       : num  -86.6 -86.8 -86.2 -86.6 -86.3 ...
##  $ CBSA      : chr  "26620" "13820" "33860" "26620" ...
##  $ NMCBSA    : chr  "Huntsville, AL" "Birmingham-Hoover, AL" "Montgomery, AL" "Huntsville, AL" ...
##  $ CBSATYPE  : int  1 1 1 1 1 1 1 2 1 1 ...
##  $ CSA       : chr  "290" "142" "388" "290" ...
##  $ NMCSA     : chr  "Huntsville-Decatur, AL" "Birmingham-Hoover-Talladega, AL" "Montgomery-Selma-Ale
##  $ NECTA     : chr  "N" "N" "N" "N" ...
##  $ NMNECTA   : chr  "N" "N" "N" "N" ...
##  $ CD        : chr  "0105" "0107" "0102" "0105" ...
##  $ SLDL      : chr  "01019" "01055" "01074" "01006" ...
##  $ SLDU      : chr  "01007" "01018" "01025" "01002" ...
##  $ SCHOOLYEAR: chr  "2020-2021" "2020-2021" "2020-2021" "2020-2021" ...
```

```
# From the output, we can see that R stores the states as characters.
school_loc$STATE = factor(school_loc$STATE)
# Let's check again. You can see that STATE is now stored as factors in R.
str(school_loc)
```

```
## 'data.frame':    7012 obs. of  26 variables:
##  $ X         : num  -86.6 -86.8 -86.2 -86.6 -86.3 ...
##  $ Y         : num  34.8 33.5 32.4 34.7 32.4 ...
##  $ OBJECTID  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ UNITID    : int  100654 100663 100690 100706 100724 100733 100751 100760 100812 100830 ...
##  $ NAME      : chr  "Alabama A & M University" "University of Alabama at Birmingham" "Amridge Univers
##  $ STREET    : chr  "4900 Meridian Street" "Administration Bldg Suite 1070" "1200 Taylor Rd" "301 Spa
##  $ CITY      : chr  "Normal" "Birmingham" "Montgomery" "Huntsville" ...
##  $ STATE     : Factor w/ 59 levels "AK","AL","AR",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ ZIP       : chr  "35762" "35294-0110" "36117-3553" "35899" ...
##  $ STFIP     : chr  "01" "01" "01" "01" ...
##  $ CNTY      : chr  "01089" "01073" "01101" "01089" ...
##  $ NMCNTY    : chr  "Madison County" "Jefferson County" "Montgomery County" "Madison County" ...
##  $ LOCALE    : chr  "12" "12" "12" "12" ...
##  $ LAT       : num  34.8 33.5 32.4 34.7 32.4 ...
##  $ LON       : num  -86.6 -86.8 -86.2 -86.6 -86.3 ...
##  $ CBSA      : chr  "26620" "13820" "33860" "26620" ...
##  $ NMCBSA    : chr  "Huntsville, AL" "Birmingham-Hoover, AL" "Montgomery, AL" "Huntsville, AL" ...
##  $ CBSATYPE  : int  1 1 1 1 1 1 1 2 1 1 ...
##  $ CSA       : chr  "290" "142" "388" "290" ...
##  $ NMCSA     : chr  "Huntsville-Decatur, AL" "Birmingham-Hoover-Talladega, AL" "Montgomery-Selma-Ale
##  $ NECTA     : chr  "N" "N" "N" "N" ...
##  $ NMNECTA   : chr  "N" "N" "N" "N" ...
##  $ CD        : chr  "0105" "0107" "0102" "0105" ...
##  $ SLDL      : chr  "01019" "01055" "01074" "01006" ...
##  $ SLDU      : chr  "01007" "01018" "01025" "01002" ...
##  $ SCHOOLYEAR: chr  "2020-2021" "2020-2021" "2020-2021" "2020-2021" ...
```

```
summary(school_loc)
```

```
##        X                  Y             OBJECTID        UNITID
##  Min.   :-170.74   Min.   :-14.32   Min.   :   1   Min.   :  100654
##  1st Qu.: -97.46   1st Qu.: 33.82   1st Qu.:1754   1st Qu.:  175401
##  Median : -86.14   Median : 38.55   Median :3506   Median :  232226
##  Mean   : -90.36   Mean   : 37.14   Mean   :3506   Mean   : 2513257
##  3rd Qu.: -78.82   3rd Qu.: 41.20   3rd Qu.:5259   3rd Qu.:  458147
##  Max.   : 171.38   Max.   : 71.32   Max.   :7012   Max.   :49576723
##
##      NAME              STREET             CITY               STATE
##  Length:7012        Length:7012        Length:7012        CA     : 752
##  Class :character   Class :character   Class :character   NY     : 463
##  Mode  :character   Mode  :character   Mode  :character   TX     : 453
##                                                           FL     : 420
##                                                           PA     : 378
##                                                           OH     : 298
##                                                           (Other):4248
##      ZIP               STFIP               CNTY              NMCNTY
##  Length:7012        Length:7012        Length:7012        Length:7012
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##     LOCALE               LAT              LON               CBSA
##  Length:7012        Min.   :-14.32   Min.   :-170.74   Length:7012
##  Class :character   1st Qu.: 33.82   1st Qu.: -97.46   Class :character
##  Mode  :character   Median : 38.55   Median : -86.14   Mode  :character
##                     Mean   : 37.14   Mean   : -90.36
##                       3rd Qu.: 41.20   3rd Qu.: -78.82
##                       Max.   : 71.32   Max.   : 171.38
##
##     NMCBSA              CBSATYPE         CSA                NMCSA
##  Length:7012        Min.   :0.000   Length:7012        Length:7012
##  Class :character   1st Qu.:1.000   Class :character   Class :character
##  Mode  :character   Median :1.000   Mode  :character   Mode  :character
##                     Mean   :1.058
##                     3rd Qu.:1.000
##                     Max.   :2.000
##
##     NECTA              NMNECTA              CD                 SLDL
##  Length:7012        Length:7012        Length:7012        Length:7012
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##      SLDU            SCHOOLYEAR
##  Length:7012        Length:7012
##  Class :character   Class :character
##  Mode  :character   Mode  :character
##
##
##
##
```

Notice the **summary()** function handles factors differently to numbers (and strings), the occurrence counts for each value is often more useful information.

The function **table()** is also useful for viewing the observations.

```
table(school_loc$STATE)
```

```
##
##  AK  AL  AR  AS  AZ  CA  CO  CT  DC  DE  FL  FM  GA  GU  HI  IA  ID  IL  IN  KS
##  10  90  94   1 131 752 104  79  28  22 420   1 183   3  25  81  41 270 138  80
##  KY  LA  MA  MD  ME  MH  MI  MN  MO  MP  MS  MT  NC  ND  NE  NH  NJ  NM  NV  NY
##  91 123 159  87  39   1 197 116 164   1  61  34 181  28  43  38 173  50  41 463
##  OH  OK  OR  PA  PR  PW  RI  SC  SD  TN  TX  UT  VA  VI  VT  WA  WI  WV  WY
## 298 110  80 378 175   1  23 101  29 165 453  73 169   2  23 110  96  73  10
```

In-class exercises 4.1:

1. Load the worldTFR dataset.
2. Get the structure of the dataframe.
3. Get the summary statistics of the dataframe.
4. What data types are each variable stored as?
5. Convert variables to factor as appropriate.

2. Probability distributions in R

Every distribution has four associated functions whose prefix indicates the type of function and the suffix indicates the distribution. In our examples, we will focus on the normal (Gaussian) distribution. We will not cover the mathematical concepts in depth for normal distribution. You will learn more about it in both QPM I and QPM II.

The four normal distribution functions are:

- dnorm: density function of the normal distribution
- pnorm: cumulative density function of the normal distribution
- qnorm: quantile function of the normal distribution
- rnorm: random sampling from the normal distribution

2.1 dnorm

```
# ?dnorm
dnorm(0:10, mean = 0, sd = 1, log = FALSE)
```

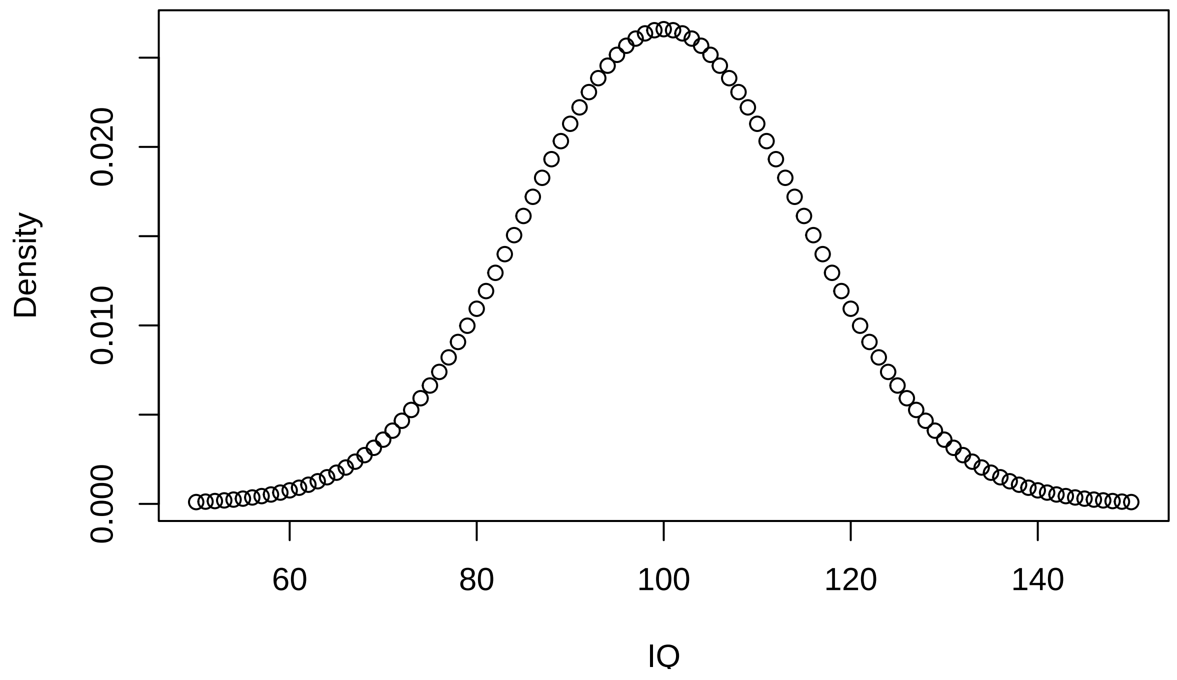
```
##  [1] 3.989423e-01 2.419707e-01 5.399097e-02 4.431848e-03 1.338302e-04
##  [6] 1.486720e-06 6.075883e-09 9.134720e-12 5.052271e-15 1.027977e-18
## [11] 7.694599e-23
```

Using the density, it is possible to determine the probabilities of events.

For example, you may wonder: What is the likelihood that a person has an IQ of exactly 140?. In this case, you would need to retrieve the density of the IQ distribution at value 140. The IQ distribution can be modeled with a mean of 100 and a standard deviation of 15. The corresponding density is:

```
sample.range <- 50:150
iq.mean <- 100
iq.sd <- 15

# This returns the density, NOT the values for IQ
iq.dist <- dnorm(sample.range, mean = iq.mean, sd = iq.sd)
iq.df <- data.frame("IQ" = sample.range, "Density" = iq.dist)
plot(iq.df)
```



From these data, we can now answer the initial question as well as additional questions. Let's write a function that returns a percentage value when given a density.

```r
pp <- function(x) {
    print(paste0(round(x * 100, 3), "%"))
}

# ?paste0

# likelihood of IQ == 140?
pp(iq.df$Density[iq.df$IQ == 140])
```

```
## [1] "0.076%"
```

```r
# likelihood of IQ >= 140?
pp(sum(iq.df$Density[iq.df$IQ >= 140]))
```

```
## [1] "0.384%"
```

```r
# likelihood of 50 < IQ <= 90?
pp(sum(iq.df$Density[iq.df$IQ <= 90]))
```

```
## [1] "26.284%"
```

```r
# What does this code do?
pp(sum(iq.df$Density[iq.df$IQ <= 100]) + sum(iq.df$Density[iq.df$IQ > 100]))
```

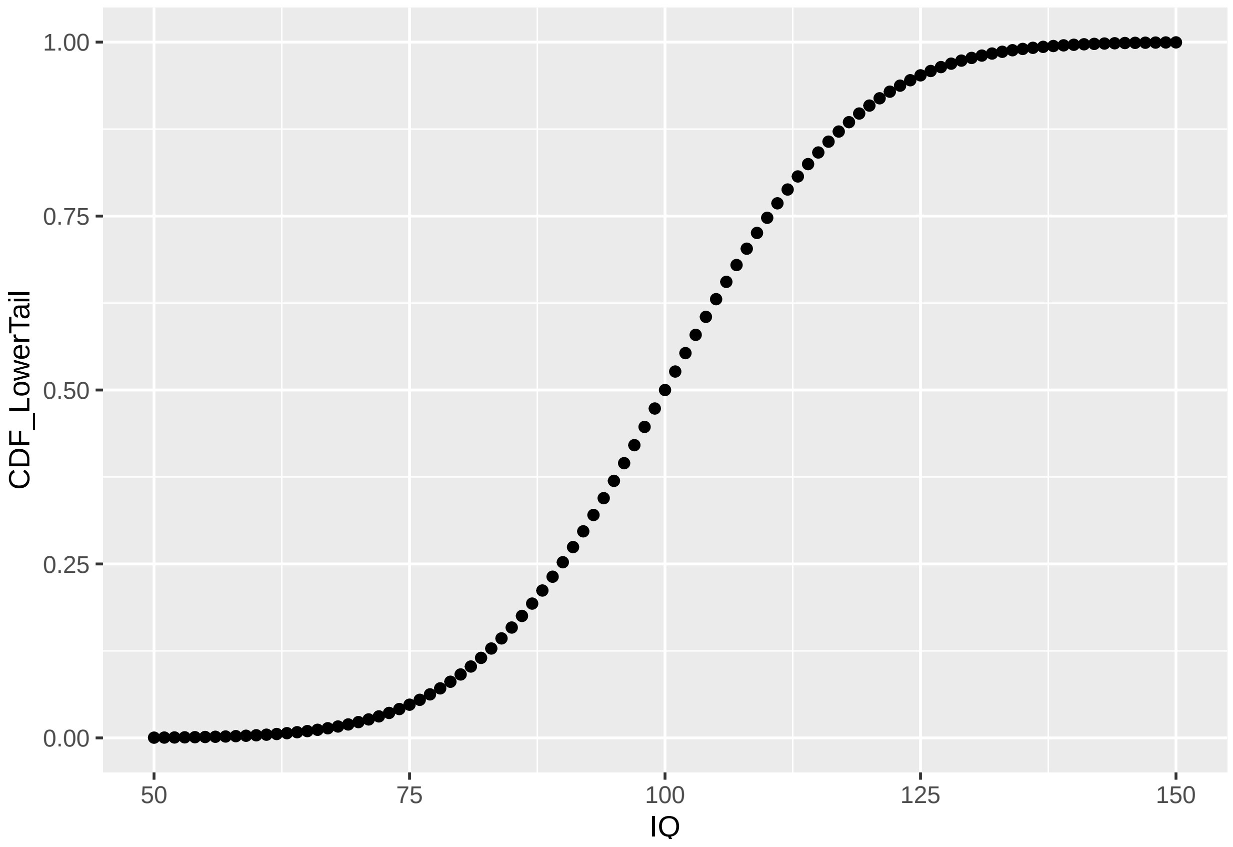
```
## [1] "99.924%"
```

2.2 pnorm

The cumulative density (CDF) function is a monotonically increasing function as it integrates over densities. To get an intuition of the CDF, let's create a plot for the IQ data:

```
cdf <- pnorm(sample.range, iq.mean, iq.sd)
iq.df <- cbind(iq.df, "CDF_LowerTail" = cdf)
# Can you explain what this code above is doing?

library(ggplot2)
ggplot(iq.df, aes(x = IQ, y = CDF_LowerTail)) + geom_point()
```



As we can see, the depicted CDF shows the probability of having an IQ less or equal to a given value. This is because pnorm computes the lower tail by default, i.e. $P[X <= x]$, as we see in the help pages. Using this knowledge, we can obtain answers to some of our previous questions in a slightly different manner.

```
# likelihood of IQ <= 140?
# notice here we using == signs
pp(iq.df$CDF_LowerTail[iq.df$IQ == 140])
```

```
## [1] "99.617%"
```

```
# likelihood of 50 < IQ <= 90?
pp(iq.df$CDF_LowerTail[iq.df$IQ == 90])
```

```
## [1] "25.249%"
```

```r
# set lower.tail to FALSE to obtain P[X >= x]
cdf <- pnorm(sample.range, iq.mean, iq.sd, lower.tail = FALSE)
iq.df <- cbind(iq.df, "CDF_UpperTail" = cdf)
# Probability for IQ >= 140? same value as before using dnorm!
pp(iq.df$CDF_UpperTail[iq.df$IQ == 140])
```

```
## [1] "0.383%"
```

Note that the results from pnorm are the same as those obtained from manually summing up the probabilities obtained via dnorm. Moreover, by setting **lower.tail = FALSE**, dnorm can be used to directly compute **p-values**, which measure how the likelihood of an observation that is at least as extreme as the obtained one.

It is important to remember that **pnorm** does NOT provide the PDF but the **CDF**.

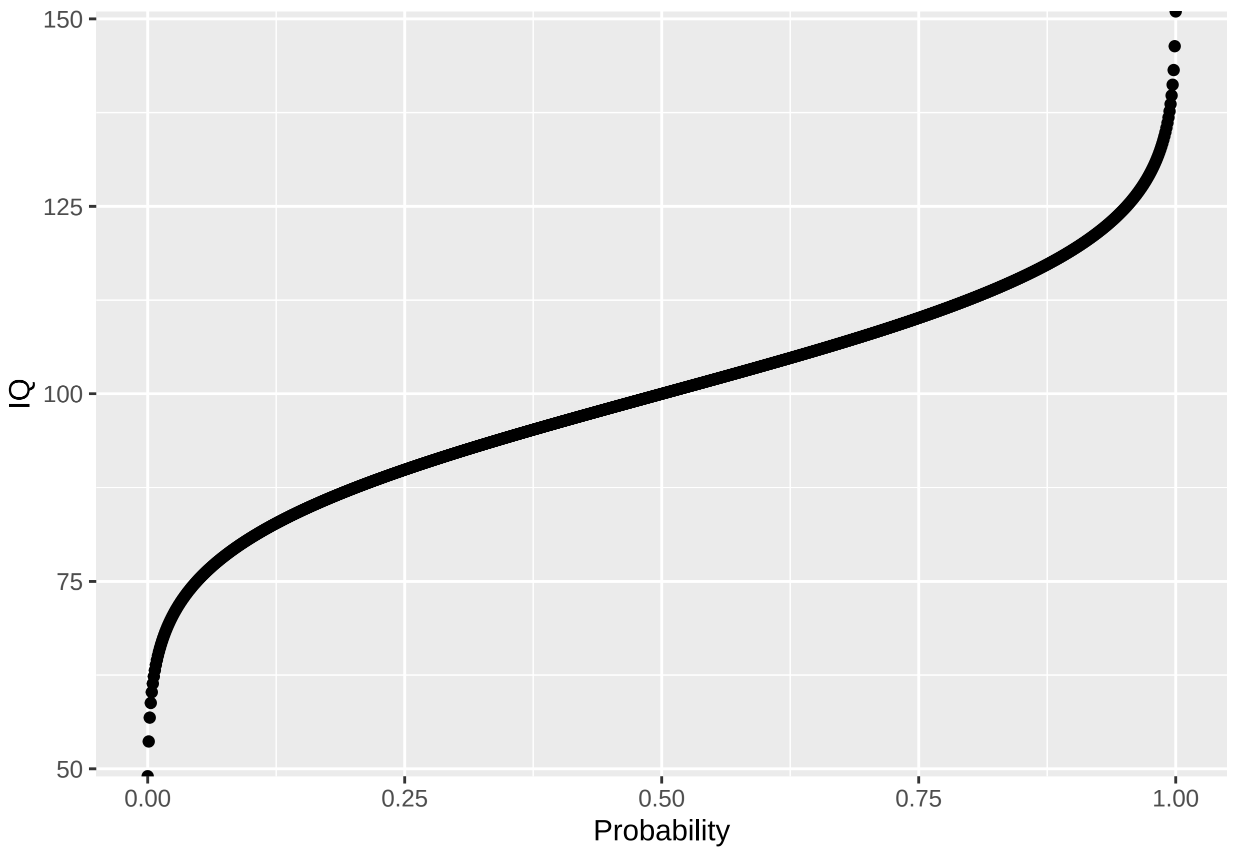
2.3 qnorm

The quantile function is simply the inverse of the cumulative density function (iCDF). Thus, the quantile function maps from probabilities to values.

Let's take a look at the quantile function for $P[X <= x]$:

```r
# input to qnorm is a vector of probabilities
prob.range <- seq(0, 1, 0.001)
# ?seq

icdf.df <- data.frame("Probability" = prob.range, "IQ" = qnorm(prob.range, iq.mean, iq.sd))
# View(icdf.df)
ggplot(icdf.df, aes(x = Probability, y = IQ)) + geom_point()
```



As illustrated with the plot, given a specific probablity, you can match it back to IQ values.

```r
# Using the quantile function, we can answer quantile-related questions:
# what is the 25th IQ percentile? (Q1)
print(icdf.df$IQ[icdf.df$Probability == 0.25])
```

```
## [1] 89.88265
```

```r
# What is the 50th IQ percentile? (median or Q2)
# It's also the mean in normal distributions.
print(icdf.df$IQ[icdf.df$Probability == 0.5])
```

```
## [1] 100
```

```r
# what is the 75th IQ percentile? (Q3)
print(icdf.df$IQ[icdf.df$Probability == 0.75])
```

```
## [1] 110.1173
```

```r
# note: this is the same results as from the quantile function
quantile(icdf.df$IQ)
```

```
##        0%       25%       50%       75%      100%
##      -Inf  89.88265 100.00000 110.11735       Inf
```

```r
# what is the 90th IQ percentile?
print(icdf.df$IQ[icdf.df$Probability == 0.9])
```

```
## [1] 119.2233
```

2.4 rnorm

When you want to draw random samples from the normal distribution, you can use **rnorm.**

For example, we could use **rnorm** to simulate random samples from the IQ distribution.

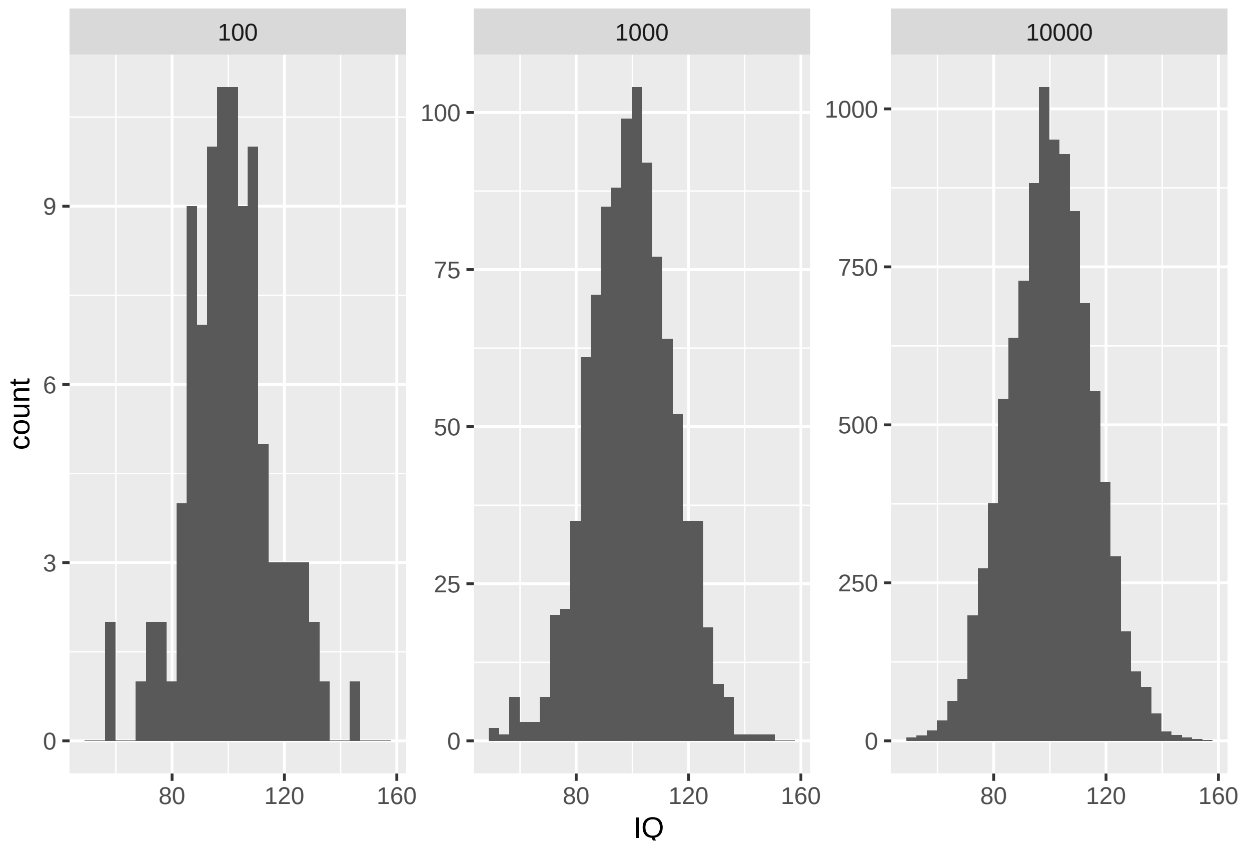
```r
# fix random seed for reproducibility
set.seed(1234)

# ?rnorm
obs <- rnorm(n=100, iq.mean, iq.sd)
mean(obs)
```

```
## [1] 97.64857
```

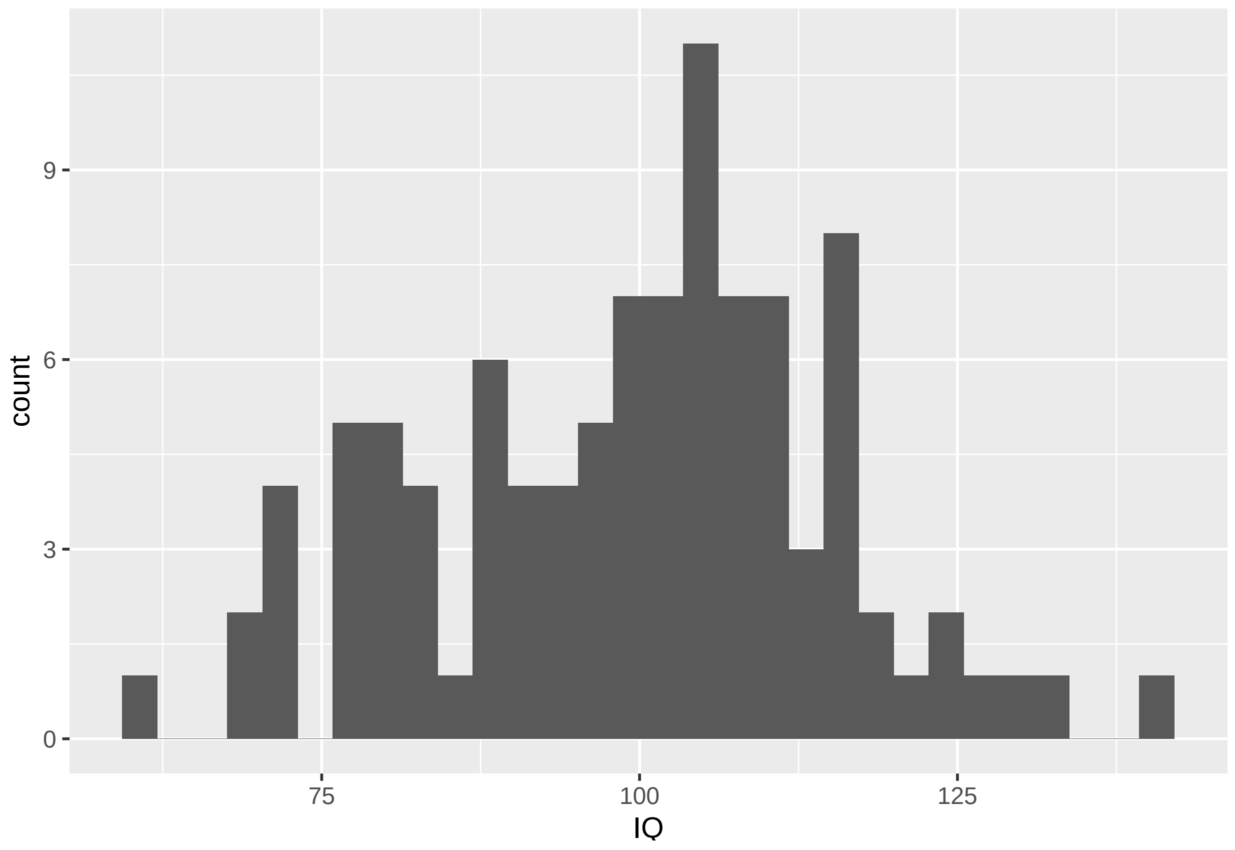
```r
## More Advanced Content:
## Don't worry if the code below does not totally make sense so far.
# law of large numbers: mean will approach expected value for large N
n.samples <- c(100, 1000, 10000)
my.df <- do.call(rbind, lapply(n.samples,
                               function(x) data.frame("SampleSize" = x,
                                                      "IQ" = rnorm(x, iq.mean, iq.sd))))
# show one facet per random sample of a given size
ggplot() + geom_histogram(data = my.df, aes(x = IQ)) + facet_wrap(.~SampleSize, scales = "free_y")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



```r
# note: we can also implement our own sampler using the densities
my.sample <- sample(iq.df$IQ, 100, prob = iq.df$Density, replace = TRUE)
my.sample.df <- data.frame("IQ" = my.sample)
ggplot(my.sample.df, aes(x = IQ)) + geom_histogram()
```

```
## ‘stat_bin()‘ using ‘bins = 30‘. Pick better value with ‘binwidth‘.
```



Note that we called **set.seed()** in order to ensure that the random number generator always generates the same sequence of numbers for reproducibility.

Of the four functions dealing with distributions, **dnorm** is the most important one. This is because the values from pnorm, qnorm, and rnorm are based on dnorm. Still, pnorm, qnorm, and rnorm are very useful convenience functions when dealing with the normal distribution. If you would like to learn about the corresponding functions for the other distributions, you can simply call **?distribtuion** to obtain more information.

In-class exercises 4.2:

Please explain what the following code does:

1.

```
pnorm(0)
```

```
## [1] 0.5
```

2.

```
pnorm(1.96, lower.tail=TRUE)
```

```
## [1] 0.9750021
```

3.

```
dnorm(0:10, mean = 0, sd = 1, log = FALSE)
```

```
##  [1] 3.989423e-01 2.419707e-01 5.399097e-02 4.431848e-03 1.338302e-04
##  [6] 1.486720e-06 6.075883e-09 9.134720e-12 5.052271e-15 1.027977e-18
## [11] 7.694599e-23
```

4.

```
qnorm(0.025)
```

```
## [1] -1.959964
```

5.

```
qnorm(0.975)
```

```
## [1] 1.959964
```

6.

```
qnorm(.975, 20, 1.65)
```

```
## [1] 23.23394
```

7.

```
rnorm(1:100, 0.05, 0.5)
```

```
##   [1]  0.630202402  0.200244077 -0.041661796 -0.619255896 -0.171655090
##   [6] -0.548189997 -0.357417393 -0.780209069  0.334189250  0.873999745
##  [11]  0.550322871  0.505605033  0.509537475 -0.902918656  0.608223702
##  [16]  1.267440834 -0.549784203  0.505663651  0.304962890  0.003464179
##  [21]  0.304634865 -0.206496016 -0.804995286  0.147167435  0.289501995
##  [26]  0.307106626 -1.225669337  0.659304733  1.177762821  0.370641113
##  [31] -0.644392999  0.626746019 -0.429274926  0.195124490  0.583880776
##  [36]  0.486160131  0.073120032 -0.032888358  0.175377744  0.433372713
##  [41] -0.183316964  0.513251126 -0.055664380  0.233197839 -0.993092193
##  [46]  0.363752653  1.371728853  0.153865202  0.330738141  0.355462507
##  [51] -0.211380867  1.260925196  0.108324409  0.022808358 -0.326517906
##  [56] -0.152991901 -0.836471104  0.077583997  0.147465141 -0.412382231
##  [61]  0.161729711 -0.421144270  0.108192011  0.469849522  0.039339958
##  [66] -0.362836624  0.411335236  0.301954737  0.117009396  1.724753892
##  [71] -0.093030941  0.953623782 -0.274619265 -0.363726050  0.217760138
##  [76] -0.608014567  0.064351334 -0.478454447 -0.063833853  0.632173601
##  [81]  1.540155188 -0.083237247 -0.013980501 -0.936736106  0.285844995
##  [86] -0.074993206  0.132686669 -0.524698302  0.012251520  0.496640518
##  [91]  0.241227855 -0.902910098 -0.411409724  0.703613833  0.372825863
##  [96] -0.166697256 -0.044302757 -0.652970998  0.293964536 -0.239699212
```

3. Descriptive Statistics

3.1 Name Commands

- names() – It works on matrix or data frame objects.

```
names(school_loc)
```

```
##  [1] "X"            "Y"            "OBJECTID"     "UNITID"       "NAME"
##  [6] "STREET"       "CITY"         "STATE"        "ZIP"          "STFIP"
## [11] "CNTY"         "NMCNTY"       "LOCALE"       "LAT"          "LON"
## [16] "CBSA"         "NMCBSA"       "CBSATYPE"     "CSA"          "NMCSA"
## [21] "NECTA"        "NMNECTA"      "CD"           "SLDL"         "SLDU"
## [26] "SCHOOLYEAR"
```

- rownames() – It works on matrix or data frame objects and is used to give names to rows.

```
rownames(school_loc)
```

```
##    [1] "1"    "2"    "3"    "4"    "5"    "6"    "7"    "8"    "9"    "10"
##   [11] "11"   "12"   "13"   "14"   "15"   "16"   "17"   "18"   "19"   "20"
##   [21] "21"   "22"   "23"   "24"   "25"   "26"   "27"   "28"   "29"   "30"
##   [31] "31"   "32"   "33"   "34"   "35"   "36"   "37"   "38"   "39"   "40"
##   [41] "41"   "42"   "43"   "44"   "45"   "46"   "47"   "48"   "49"   "50"
##   [51] "51"   "52"   "53"   "54"   "55"   "56"   "57"   "58"   "59"   "60"
##   [61] "61"   "62"   "63"   "64"   "65"   "66"   "67"   "68"   "69"   "70"
##   [71] "71"   "72"   "73"   "74"   "75"   "76"   "77"   "78"   "79"   "80"
##   [81] "81"   "82"   "83"   "84"   "85"   "86"   "87"   "88"   "89"   "90"
##   [91] "91"   "92"   "93"   "94"   "95"   "96"   "97"   "98"   "99"   "100"
...
```

- colnames() – It works on matrix or data frame objects and is used to give names to columns.

```
colnames(school_loc)
```

```
##  [1] "X"            "Y"            "OBJECTID"     "UNITID"       "NAME"
##  [6] "STREET"       "CITY"         "STATE"        "ZIP"          "STFIP"
## [11] "CNTY"         "NMCNTY"       "LOCALE"       "LAT"          "LON"
## [16] "CBSA"         "NMCBSA"       "CBSATYPE"     "CSA"          "NMCSA"
## [21] "NECTA"        "NMNECTA"      "CD"           "SLDL"         "SLDU"
## [26] "SCHOOLYEAR"
```

- dimnames() – Gets row and column names for matrix or data frame objects, that is, it is used to see dimensions of the data frame.

```
dimnames(school_loc)[1]
```

```
## [[1]]
##    [1] "1"    "2"    "3"    "4"    "5"    "6"    "7"    "8"    "9"    "10"
##   [11] "11"   "12"   "13"   "14"   "15"   "16"   "17"   "18"   "19"   "20"
##  [21] "21"   "22"   "23"   "24"   "25"   "26"   "27"   "28"   "29"   "30"
##  [31] "31"   "32"   "33"   "34"   "35"   "36"   "37"   "38"   "39"   "40"
##  [41] "41"   "42"   "43"   "44"   "45"   "46"   "47"   "48"   "49"   "50"
##  [51] "51"   "52"   "53"   "54"   "55"   "56"   "57"   "58"   "59"   "60"
##  [61] "61"   "62"   "63"   "64"   "65"   "66"   "67"   "68"   "69"   "70"
##  [71] "71"   "72"   "73"   "74"   "75"   "76"   "77"   "78"   "79"   "80"
##  [81] "81"   "82"   "83"   "84"   "85"   "86"   "87"   "88"   "89"   "90"
...
```

```
dimnames(school_loc)[2]
```

```
## [[1]]
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR"
```

3.2 Summarizing Commands

- max(x, na.rm = FALSE) – It shows the maximum value. By default, NA values are not removed. NA is considered the largest unless na.rm=true is used.
- min(x, na.rm = FALSE) – Shows minimum value in a vector. If there are na values, NA is returned unless na.rm=true is used.
- length(x) – Gives length of the vector and includes na values. Na.rm=instruction does not work with this command.
- sum(x, na.rm = FALSE) – Shows the sum of the vector elements.
- mean(x, na.rm = FALSE) – We obtain an arithmetic mean with this.
- median(x, na.rm = FALSE) – Shows the median value of the vector.
- sd(x, na.rm = FALSE) – Shows the standard deviation.
- var(x, na.rm = FALSE) – Shows the variance.
- mad(x, na.rm = FALSE) – Shows the median absolute deviation.
- log(dataset) – Shows log value for each element.
- summary(dataset) – We have seen how it shows a summary of dataset like maximum value, minimum value, mean, etc.
- quantile() – Shows the quantiles by default—the 0%, 25%, 50%, 75%, and 100% quantiles. You can select other quantiles also.

```
x <- sample(1:100000, 2000)
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE, names = TRUE)
```

```
##      0%     25%     50%     75%    100%
##    81.0 25566.0 50372.5 75887.0 99996.0
```

We will do more exercises where you would be asked to find these summarizing statistics given a dataframe in the afternoon session.

3.3 Accessing Elements from Summary Statistics

You might want to access elements from the summary statistics sometimes. We will use the built-in dataset longley again to show some examples.

```r
df <- longley
s <- summary(df)
s
```

```
##   GNP.deflator         GNP          Unemployed     Armed.Forces  
##  Min.   : 83.00   Min.   :234.3   Min.   :187.0   Min.   :145.6  
##  1st Qu.: 94.53   1st Qu.:317.9   1st Qu.:234.8   1st Qu.:229.8  
##  Median :100.60   Median :381.4   Median :314.4   Median :271.8  
##  Mean   :101.68   Mean   :387.7   Mean   :319.3   Mean   :260.7  
##  3rd Qu.:111.25   3rd Qu.:454.1   3rd Qu.:384.2   3rd Qu.:306.1  
##  Max.   :116.90   Max.   :554.9   Max.   :480.6   Max.   :359.4  
##    Population         Year         Employed    
##  Min.   :107.6   Min.   :1947   Min.   :60.17  
##  1st Qu.:111.8   1st Qu.:1951   1st Qu.:62.71  
##  Median :116.8   Median :1954   Median :65.50  
##  Mean   :117.4   Mean   :1954   Mean   :65.32  
##  3rd Qu.:122.3   3rd Qu.:1958   3rd Qu.:68.29  
##  Max.   :130.1   Max.   :1962   Max.   :70.55
```

```r
# You can simply use indices to access each element like this:
s[1]
```

```
## [1] "Min.   : 83.00  "
```

When we go into linear regression in QPM I, accessing elements of list-like objects would become more useful. For example, we can fit a linear model using the lm() function where the independent variable is population and the dependent variable is the number of unemployed.

```r
lm <- lm(Unemployed ~ Population, data = df)
```

Let’s check the structure of the object:

```r
str(lm)
```

```
## List of 12
##  $ coefficients : Named num [1:2] -763.66 9.22
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Population"
##  $ residuals    : Named num [1:16] 6.8 -5.74 119.43 75.67 -60.1 ...
##   ..- attr(*, "names")= chr [1:16] "1947" "1948" "1949" "1950" ...
##  $ effects      : Named num [1:16] -1277.3 248.5 119.3 75.3 -60.7 ...
##   ..- attr(*, "names")= chr [1:16] "(Intercept)" "Population" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:16] 229 238 249 259 270 ...
##   ..- attr(*, "names")= chr [1:16] "1947" "1948" "1949" "1950" ...
...
```

You can use the summary() function again on lm.

```r
summary(lm)
```

```
##
## Call:
## lm(formula = Unemployed ~ Population, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -110.842  -50.308   -2.621   53.460  119.434
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
...
```

If you want to access elements from the summary statistics, you can use dollar signs and brackets, just like we did on Day 2.

```
lm$coefficients
```

```
## (Intercept)  Population
## -763.664561    9.222951
```

```
lm$residuals
```

```
##         1947         1948         1949         1950         1951         1952
##    6.8012381   -5.7430639  119.4335489   75.6718174  -60.0976845  -87.8191111
##         1953         1954         1955         1956         1957         1958
## -110.8417739   49.5824061  -28.5992238  -49.2133159  -53.5937853  107.0256733
##         1959         1960         1961         1962
##    7.1659745    0.5016264   65.0918159  -35.3661421
```

```
lm$coefficients[1]
```

```
## (Intercept)
##   -763.6646
```

```
lm$coefficients[[1]]
```

```
## [1] -763.6646
```

In-class exercises 4.3:

We will continue using the TFR dataset.

1. Omit all NA's in the dataframe.
2. Run a linear regression of ChildBearing on TFR.
3. Get the coefficients.
4. What are the column names and row names?
5. What is the max and min of Years?
6. What is the average Life Expectancy at Birth (LifeExpB)?
7. What is the median Life Expectancy at Birth (LifeExpB)?
8. What is the 25% and 75% quantile of Life Expectancy at Birth (LifeExpB)?

```r
TFR <- read.csv("worldTFR.csv")

# 4.
colnames(TFR)
rownames(TFR)

# 5.
max(TFR$Year)
min(TFR$Year)

# 6.
mean(TFR$LifeExpB)

#7.
median(TFR$LifeExpB)

# 8.
quantile(TFR$LifeExpB)
```