

R Setup and Notes

Sam Gifford

2024-09-04

Downloading R

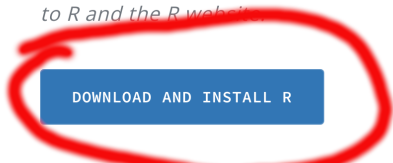
<https://posit.co/download/rstudio-desktop/>

First download R. You'll want to follow the links based on your specific system. The following screenshots show how this works for Windows, though you'll follow similar instructions for a mac. If using a chromebook you'll need to see the specific instructions below. Once you have the installer downloaded, you should be able to click "next", "install", or "finish" with the default options without any issue.

1: Install R

RStudio requires R 3.6.0+. Choose a version of R that matches your computer's operating system.

R is not a Posit product. By clicking on the link below to download and install R, you are leaving the Posit website. Posit disclaims any obligations and all liability with respect to R and the R website.



2: Install RStudio

Find your operating system in the table below.

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#) ([Debian](#), [Fedora/Redhat](#), [Ubuntu](#))
- [Download R for macOS](#) ←
- [Download R for Windows](#) ←

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

R for Windows

Subdirectories:

[base](#)
[contrib](#)
[old contrib](#)
[Rtools](#)

Binaries for base distribution. This is what you want to [install R for the first time](#).

Binaries of contributed CRAN packages (for R >= 4.0.x).

Binaries of contributed CRAN packages for outdated versions of R (for R < 4.0.x).

Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.


R-4.4.1 for Windows

[Download R-4.4.1 for Windows](#) (82 megabytes, 64 bit)
[README on the Windows binary distribution](#)
[New features in this version](#)

then download and run Rstudio desktop. R can be used as a standalone with RGui (or via command line) but most people use Rstudio.

Installing Rstudio

For rstudio, use the same link as above and scroll down to directly obtain the executable or dpkg file for windows and mac OS, respectively. Similar to the base R installation, the default options will all be good. It will ask you to associate RStudio with an R installation at some point, so make sure you have installed R first. The default option should always be good to use, so you can just click “next” using all default options as before.

Installation, depending on the operating system's security policy.			
 PRODUCTS ▾ SOLUTIONS ▾ LEARN & SUPPORT ▾ EXPLORE MORE ▾ PRICING			
OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2024.09.0-375.EXE ⚙	265.55 MB	513216FE
macOS 12+	RSTUDIO-2024.09.0-375.DMG ⚙	621.00 MB	54D722FD
Ubuntu 20/Debian 11	RSTUDIO-2024.09.0-375-AMD64.DEB ⚙	203.93 MB	DB096050
Ubuntu 22/Debian 12	RSTUDIO-2024.09.0-375-AMD64.DEB ⚙	203.92 MB	111C64DB
Ubuntu 24	RSTUDIO-2024.09.0-375-AMD64.DEB ⚙	203.92 MB	111C64DB

Confirming the R installation

Just to clarify, R is the programming language, which comes with a command line interface (that is amazing for power users but otherwise no one uses) as well as an “RGui” application that allows you to run R code in

a graphical interface. Virtually everyone uses Rstudio, which is an alternative to the base RGui with much more functionality. Rstudio is an example of an IDE, or integrated development environment. The base icons for the two programs looks similar, so make sure you're launching Rstudio:

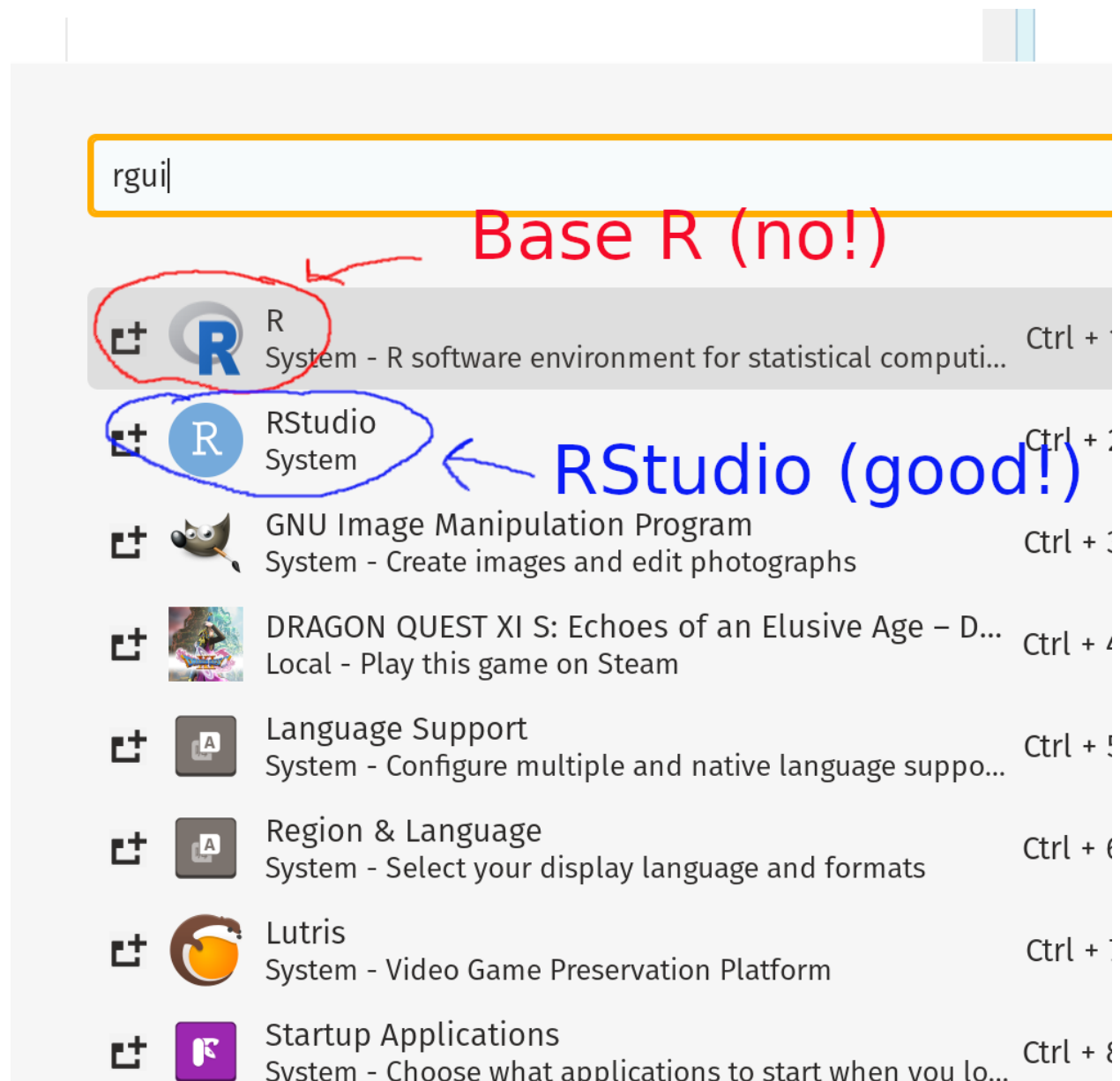
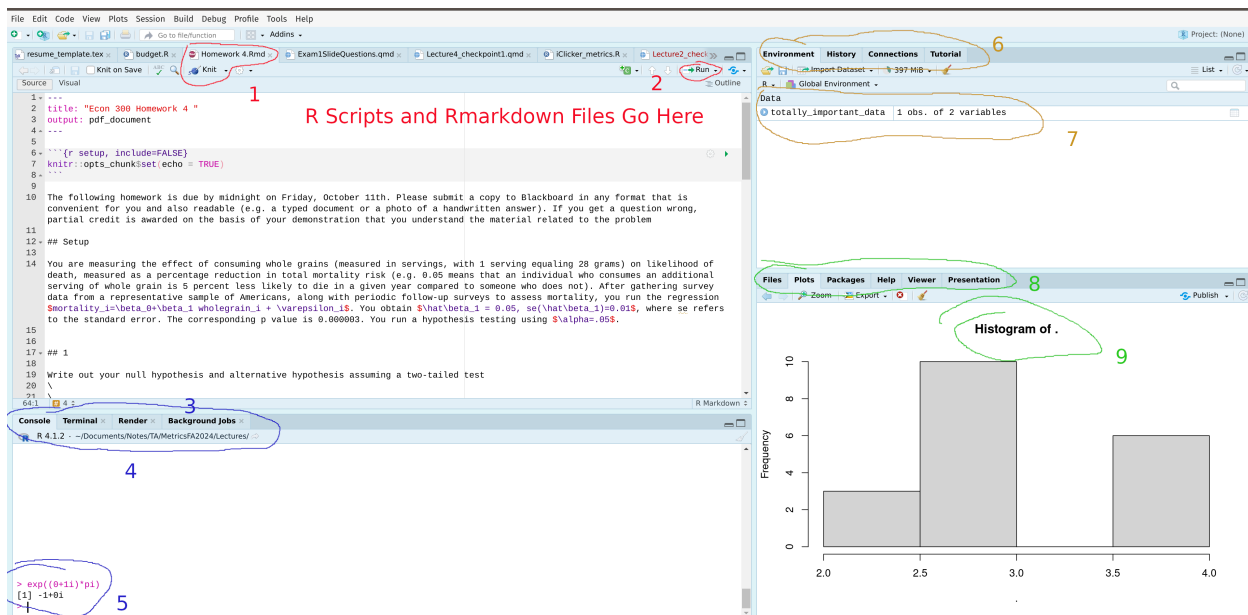


Figure 1: This will look a bit different from your system since I'm running Debian. And since some of my games library is popping up in the search.

Using Rstudio

Rstudio is broken up into 4 quadrants by default (you may be missing the upper left quadrant if you haven't opened an R file yet). A basic overview of this is given below



1. Files go in the upper left panel by default. If the file has unsaved changes it's name will be displayed in red. In general Rstudio's autosaving features are great, but you should still save frequently to avoid losing progress. If you have an rstudio file, a 'knit' option (circled) will be available, which will allow you to convert a markdown file into PDF or other portable, human-readable formats
2. You can run the entire contents of a script by pressing this button in the upper-right, or by using the "source" function. More commonly, you can run an individual line of code in a file by pressing control+enter (command+enter on a mac)
3. Your console will be in the lower left by default, where you can run scratch code that you don't need to save. Note that there are several tabs here - you also have access to a terminal (for advanced users), a render tab for rmarkdown, nad a background jobs tab (also for advanced users). When you knit a document your focus will be automatically moved from console to 'render', so if you can't run code in console please make sure you are on the correct tab
4. Directly below the console tab is your version of R and your working directory. When attempting to read in files, your base path will be this. `~` refers to your home directory, and is generally something like "C:/Users/username" but it can vary by system. You can change this with the 'setwd' function
5. This is the part of the console where you write code. Do not run important code here as it's easy to lose work this way. I normally use this to test out syntax if I'm unsure.
6. This tab contains your environment by default, which lists out every variable you have stored. You can also get this by using the "ls" function. The history tab is also available here, which includes every line of code you've run in console since starting the session (in case you need to retrieve something you previously wrote)
7. Clicking on data frames will bring up an excel-style visualization of the data set (it calls the "view" function). Once you understand how to interact with data in R you will never use this feature, as it is far too slow.
8. Most commonly your plots will appear in this panel. The file explorer is also useful if you are unsure of where your working directory is. You can also bring up help documentation for any function by typing `?functionName` in console (e.g. `?read.csv`). Quality of documentation will vary by package, but is generally a good place to start if you don't know how to call a function
9. Your plots will go here - in the upper-left is an option to export the plot to an image file if you're unsure of how to do that via code (functions like 'png' and 'pdf' will do this).

Setting up Rstudio Server for Chromebooks

If using a Chromebook you'll want to use the Linux (Debian) install using the built in terminal. The link has more information, but the following should be the barebones of what is needed. You'll run these commands within your linux terminal (penguin). The keys and file links change periodically, but these are up to date as of September 4th, 2024

```
sudo apt-get update
sudo apt-get install r-base r-base-dev
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key '95C0FAF38DB3CCAD0C080A7BDC78B2DDEABC47B7'
```

This will install R as a command line program. You then need to install rstudio server (<https://posit.co/download/rstudio-server/>). You'll need to run

```
sudo apt-get install gdebi-core
wget https://download2.rstudio.org/server/focal/amd64/rstudio-server-2024.04.2-764-amd64.deb
sudo gdebi rstudio-server-2024.04.2-764-amd64.deb
```

To launch Rstudio server use

`/usr/sbin/rstudio-server` start then open 127.0.0.1:8787 in url bar in chrome

Setting Up Rmarkdown for chrome

A tex package needs to be installed to render PDFs for rmarkdown - the following will work in a *nix environment with the first run in terminal and the second run within R

```
sudo apt-get install texlive
```

```
install.packages("tinytex")
tinytex::install_tinytex()
```

Using R

Once Rstudio is opened you'll have a console on the bottom left where you can execute R commands, and a file in the upper left panel where you can type R commands to be saved. It is recommended that commands be written in a file so that any work you perform is reproducible.

Basic Math

R supports basic math, eg

```
1+1
```

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

```
exp(2)
```

```
## [1] 7.389056
```

```
factorial(5)
```

```
## [1] 120
```

Output can be stored in variables using the <- or = operators (they are identical, but I prefer <- for style reasons).

```
x <- 5  
y <- x + 1  
print(y)
```

```
## [1] 6
```

Vectors and Objects

One key difference from R vs most languages is that it natively supports vectors. vectors can be manually specified using the c function (short for either combine or concatenate), or sequences can be generated using the : function

```
x <- c(1,2,3)  
x+1
```

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

```
y <- 1:10  
y^2
```

```
## [1] 1 4 9 16 25 36 49 64 81 100
```

R supports a wide variety of objects, eg matrices or linear models

```
A <- matrix(c(1,3,2,4),2,2)  
B <- matrix(c(1,2,3,4),2,2)  
A%*%B
```

```
##      [,1] [,2]  
## [1,] 5   11  
## [2,] 11  25
```

```
solve(A)
```

```
##      [,1] [,2]  
## [1,] -2.0  1.0  
## [2,]  1.5 -0.5
```

Working Directories and Data

Most complicated data you'll be using will need to be read in from an outside source. To do this you'll need to specify the path to the file, which is often used as a relative path, with the base path called the working directory. You can find your current working directory with `getwd()`, and set it with `setwd()`. backslashes are special characters in R ("escape characters") so you'll want to use forward slashes in addresses. You can go up a level of a directory using `..`. These are commented out below

```
#getwd()
#setwd("C:/Users")
#getwd()
#setwd("..")
#getwd()
```

Once a working directory is found, you can read in a file using `read.csv` (there are much better functions to read in files than the base R function. I use `fread` from the `data.table` package personally). Note that my working directory is not set to the above because of the quirks of this `rmarkdown` file (they're relative to where this file is stored). Below I read in a file in R, which is stored as a `data.frame` object. Some operations that can be performed with the data frame are shown below - you can preview the data by typing the variable in console, and you can get the names of the columns with the function `names`. Individual columns (which are vectors) can be accessed with `df$variableName` or `df[["variableName"]]`. The first is more common to see, but the second has the advantage of being able to have a variable passed to it (the first is called nonstandard evaluation vs the second of standard evaluation, which you typically see in programming languages)

```
df <- read.csv("../data/cps_sample.csv")
nrow(df)
```

```
## [1] 50000
```

```
ncol(df)
```

```
## [1] 9
```

```
names(df)
```

```
## [1] "year"      "statefip"  "age"       "sex"       "empstat"
## [6] "educ"      "wkswork1"  "uhrsworkly" "incwage"
```

Exploring data: numeric types

Each variable in `r` is a vector with a common class. For the CPS sample we just read in our data has 50,000 rows, so each column is a vector with 50,000 observations. We can learn more about these through various commands. Let's start with `year`

```
length(df$year)
```

```
## [1] 50000
```

```
class(df$year)
```

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

We can see that year is an integer vector. Each value is an integer (whole number), so we can summarize the data with common numeric functions. The summary function will give the “5 number summary” plus the mean

```
summary(df$year)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      2000    2000    2010    2010    2017    2017
```

We can see the data ranges from 2000 to 2017, with a mean of 2010. But are there other years in the data? We can get a count of all of our observations using the table function

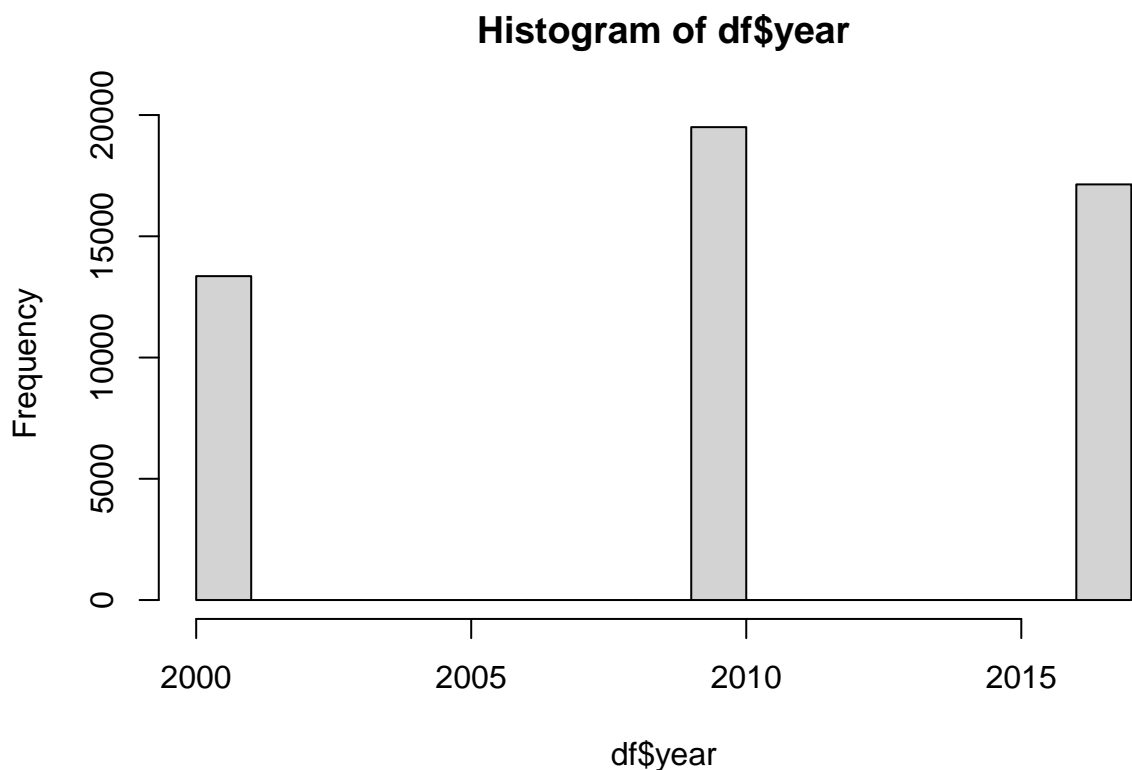
```
table(df$year)
```

```
##
## 2000 2010 2017
## 13357 19502 17141
```

Here we can see that 3 years are represented in the data, with 2010 being the most prominent.

Here we only have 3 buckets so a table is nice, but if we have more levels we can instead use a histogram to summarize the data

```
hist(df$year)
```



prettiest graph, but very fast.

Not the

Character data

The other most common type of data you'll see is character data, sometimes called strings in other languages. `statefip` (state) is an example of a variable with this class

```
class(df$statefip)
```

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

One way to look at this data is to see how many unique responses we have

```
length(unique(df$statefip))
```

```
## [1] 51
```

Here we can see our usual 51 states. Since there are relatively few observations we can run `table` on this data

```
table(df$statefip)
```

```
##
##          alabama          alaska          arizona
##          655            663            758
##          arkansas        california        colorado
##          607            4822           849
##          connecticut      delaware district of columbia
##          750            645            762
##          florida         georgia          hawaii
##          2215           974            741
##          idaho           illinois         indiana
##          708            1717           714
##          iowa            kansas          kentucky
##          742            643            629
##          louisiana       maine           maryland
##          678            581            865
##          massachusetts    michigan        minnesota
##          949            1285           910
##          mississippi      missouri        montana
##          594            699            673
##          nebraska         nevada          new hampshire
##          676            738            785
##          new jersey       new mexico      new york
##          1227           696            2294
##          north carolina    north dakota    ohio
##          1139           680            1469
##          oklahoma         oregon          pennsylvania
##          637            685            1460
##          rhode island      south carolina  south dakota
##          566            600            650
##          tennessee        texas          utah
##          677            2989           644
##          vermont          virginia        washington
##          640            977            838
##          west virginia     wisconsin      wyoming
##          598            799            708
```

Informative, but a bit hard to read. And what if we have thousands of unique responses? We can sort our table data, then take only the first or last few observations. `head` gives the first few observations (5 by default) while `tail` gives the last 5. By default sorting is done ascendingly, so we'll reverse it (with `rev`) and then take the head

```
head(rev(sort(table(df$statefip))))
```

```
##
## california      texas    new york    florida    illinois    ohio
##          4822        2989        2294        2215        1717        1469
```

Note that this does what we want of seeing the most common responses, but the actual command is heavily nested and the natural order of operation is from inside to outside. As of R 4.1, R now natively supports the pipe operator which you'll commonly see people use. Note that you'll more commonly see `%>%` used as a pipe more frequently since this was introduced many years before it was first available in base R

```
df$statefip |>
  table() |>
  sort() |>
  rev() |>
  head()
```

```
##
## california      texas    new york    florida    illinois    ohio
##          4822        2989        2294        2215        1717        1469
```

Do I have to analyze every column separately?

Here we manually looked at each column to look at its class. `r` supports loops and other constructs that allow you to do this in 1 statement. R has a builtin function called `sapply` (and `lapply`) which abstracts away from loops and let's you run this in one command. Just supply a list and a function, and it'll return that function on each element of the list. A data frame is internally a list of column vectors, so it's easy to get the classes of each

```
sapply(df,class)
```

```
##      year    statefip      age      sex    empstat      educ
## "integer" "character" "integer" "character" "character" "character"
##   wkswork1  uhrsworkly    incwage
## "integer"  "integer"    "integer"
```

As long as your function works on every column you'll get a result. `sapply` reduces your answer down to a vector which isn't always possible, so we'll use `lapply` to get the summary of each column in the dataset (note that summary applied to a character isn't very informative)

```
lapply(df,summary)
```

```
## $year
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2000   2000    2010    2010   2017    2017
```

```
##
## $statefip
##      Length      Class      Mode
##      50000 character character
##
## $age
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      18.00   30.00   40.00   39.72   49.00   64.00
##
## $sex
##      Length      Class      Mode
##      50000 character character
##
## $empstat
##      Length      Class      Mode
##      50000 character character
##
## $educ
##      Length      Class      Mode
##      50000 character character
##
## $wkswork1
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      1.00   52.00   52.00   46.95   52.00   52.00
##
## $uhrsworkly
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      1.0    40.0    40.0    39.5    40.0    99.0
##
## $incwage
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      1    16000    30000   43349   53000  1099999
```

Moments

We can calculate all of our sample statistics we've done in class using the builtin functions

```
mean(df$incwage)
```

```
## [1] 43348.96
```

```
sd(df$incwage)
```

```
## [1] 55827.43
```

Note that we can also calculate these manually. Here we make use of variable assignment to make our code a bit cleaner

```
mu <- sum(df$incwage)/length(df$incwage)
se <- (df$incwage-mu)^2
s2 <- mean(se)
s <- sqrt(s2)
mu
```

```
## [1] 43348.96
```

```
s
```

```
## [1] 55826.87
```

Note that the mean is exactly equal, but that variance is slightly off since it's using sample mean for sd. We can use the comparison operator (`==`) to check if two things are equal. It returns a boolean (true/false) vector.

```
(s/sd(df$incwage))^2 == 49999/50000
```

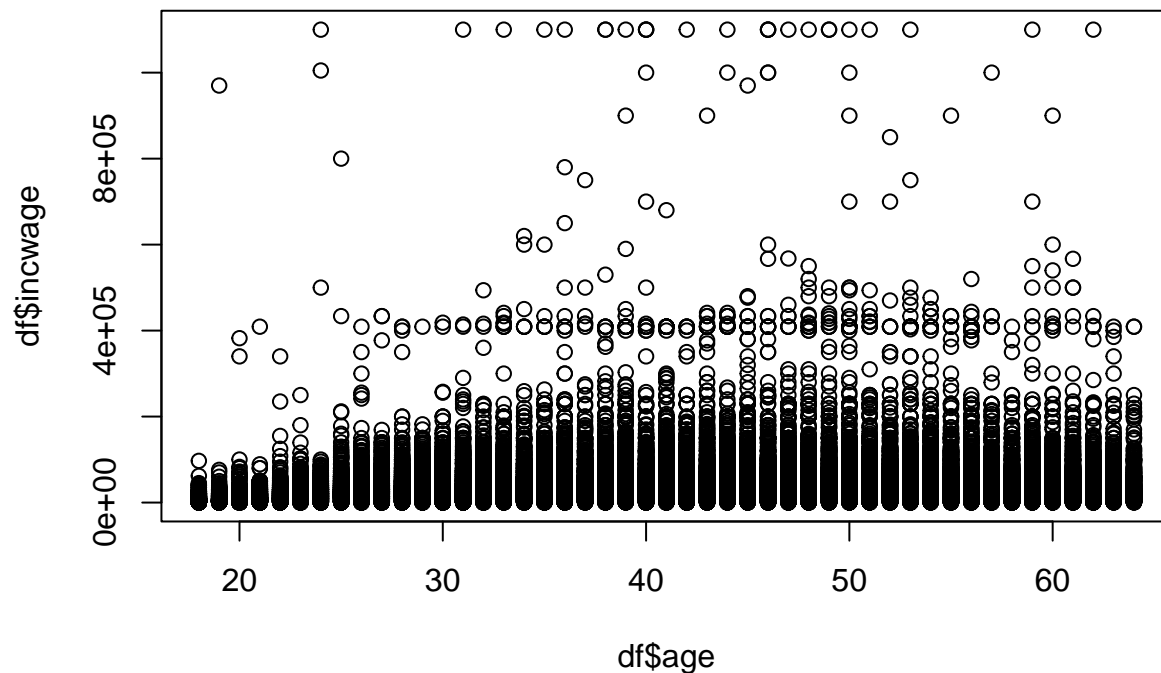
```
## [1] TRUE
```

Generating Random Numbers

Plots and Regression

We can create a basic scatterplot using the plot function

```
plot(df$age,df$incwage)
```



Generally a scatterplot using a lot of data won't work very well since there will be a lot of overlapping. Instead we can aggregate the data first. For this we can use the dplyr package. This isn't critical yet and will be deferred.

To run a linear regression on data the function `lm` is used. Here we regress income on age, ie $income_i = \beta_0 + \beta_1 age_i + \varepsilon_i$ Here we obtain $\hat{\beta}_1 = 885$

```
m <- lm(data=df, incwage ~ age)
summary(m)
```

```
##
## Call:
## lm(formula = incwage ~ age, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -64709  -23896  -10861    8825  1070563
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   8194.44     834.90   9.815  <2e-16 ***
## age           885.07      20.09  44.045  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 54780 on 49998 degrees of freedom
## Multiple R-squared:  0.03735,    Adjusted R-squared:  0.03733
## F-statistic: 1940 on 1 and 49998 DF,  p-value: < 2.2e-16
```

Subsetting and transforming data: dplyr

To aggregate we make use of a package called dplyr. We can install packages easily from within R using `install.packages`, then reference them by calling `library`. The `install.packages` line has been commented out because it only needs to be run once and I already have it installed.

You'll get some red text when loading dplyr: this isn't an error, it's just letting you know about some technical details involving namespaces.

dplyr's aggregation tends to be highly intuitive and make use of the pipe operator. You basically tell it what to do step by step. Here we are going to get the average income and number of observations for each age.

```
#install.packages('dplyr')
library(dplyr)

df_agg <- df %>%
  group_by(age) %>%
  summarise(income=mean(incwage), n=n())

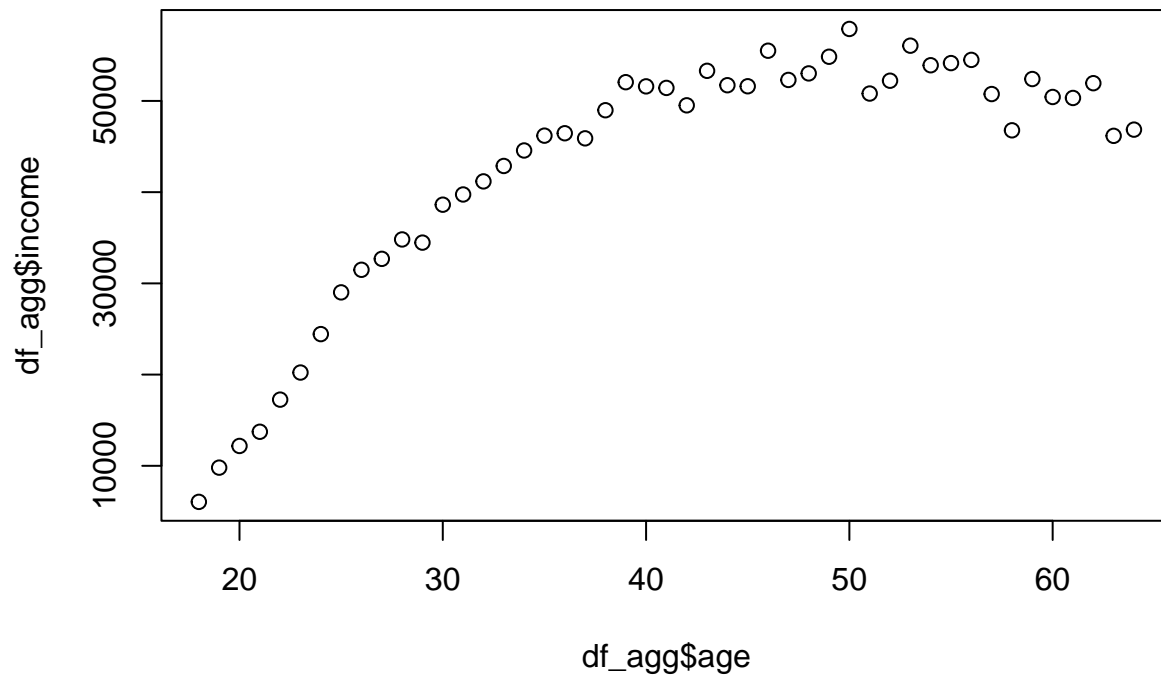
df_agg
```

```
## # A tibble: 47 x 3
##   age income      n
##   <int> <dbl> <int>
## 1    18  6049.   776
## 2    19  9802.   856
## 3    20 12177.   942
## 4    21 13734.   962
## 5    22 17255.   962
## 6    23 20227.   971
## 7    24 24439.  1056
```

```
## 8    25 29014.  1078
## 9    26 31489.  1132
## 10   27 32688.  1139
## # i 37 more rows
```

We can now make a proper scatterplot of our data

```
plot(df_agg$age,df_agg$income)
```



Packages and ggplot

To make things look nice we make use of a package called ggplot2. The syntax for ggplot2 is a bit wonky but can create awesome graphs once configured properly. After updating my R version and packages this morning this is not rendering inside my knit document, but is elsewhere so I'll work on getting this fixed

```
#install.packages('ggplot2')
library(ggplot2)

ggplot(data=df_agg, aes(x=age, y=income)) +
  geom_point() +
  geom_smooth(method="lm") +
  theme_bw() +
  ggtitle("Age vs Income, CPS Sample 2000-2017") +
  theme(plot.title = element_text(hjust = 0.5))
```

Predicting values with regression

We have our model m from before of $income_i = \beta_0 + \beta_1 age_i + \varepsilon_i$. How do we then use this data to predict \hat{income}_i ? There is a builtin function called predict that will do exactly this, and we can use dplyr to add it to our data set:

```
df <- df %>%
  mutate(predicted = predict(m,df)) %>%
  mutate(residual = incwage-predicted)
```

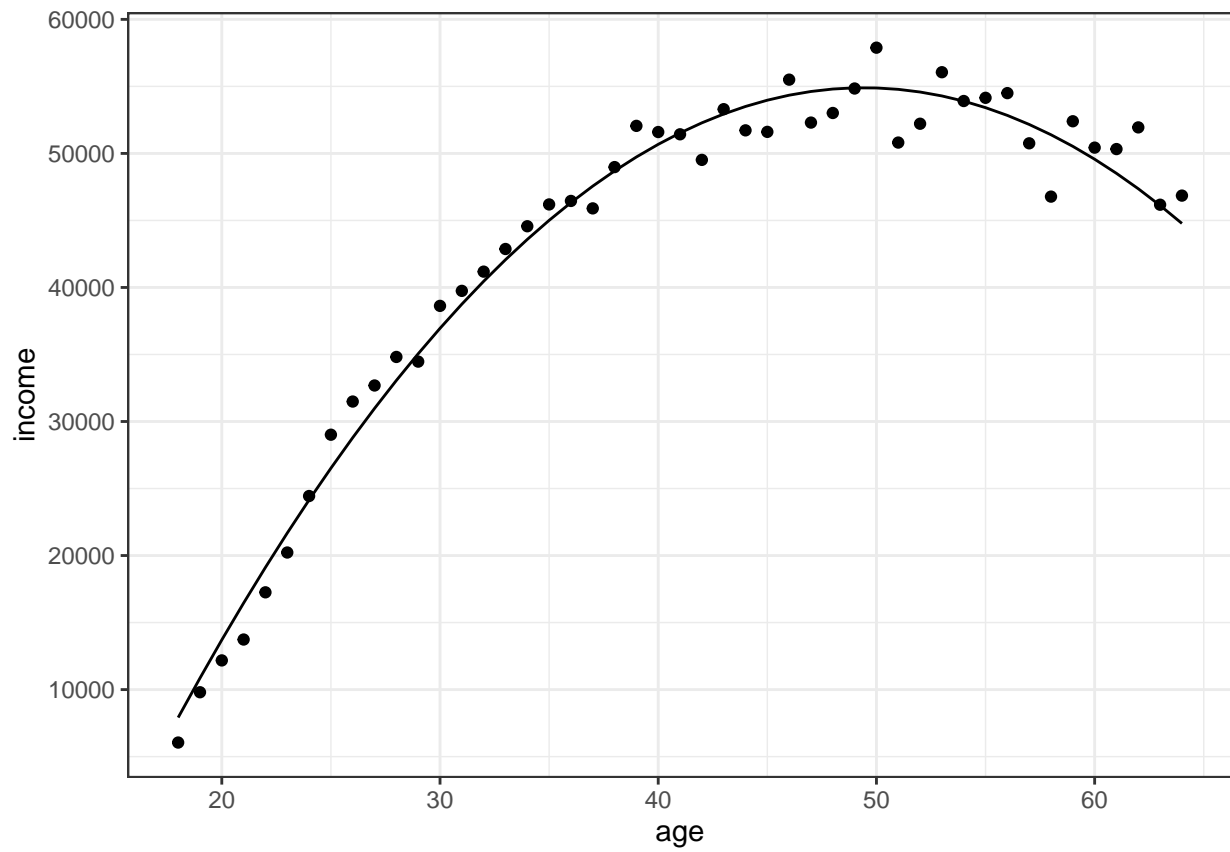
A better fit

There are tradeoffs with more complicated models, but if you wanted to fit the data above better you could add a quadratic term. First we add a new column equal to the square of age to our dataset, then we redo our model, aggregation, and plot. The last line is commented out since there

```
library(ggplot2)
df <- df %>%
  mutate(age2 = age^2)
m_quadratic <- lm(data=df, incwage ~ age + age2)
df <- df %>%
  mutate(predicted_quadratic = predict(m_quadratic,df)) %>%
  mutate(residual_quadratic = incwage-predicted_quadratic)

df_quadratic <- df %>%
  group_by(age) %>%
  summarize(income=mean(incwage), predicted=mean(predicted_quadratic))

ggplot(data=df_quadratic, aes(x=age, y=income)) +
  geom_point() +
  geom_line(aes(x=age, y=predicted)) +
  theme_bw()
```



In general if you are having trouble with something R just google it - you'll probably end up on a stackoverflow thread where your question has already been answered