# Part I
# Learn You Some R, Practically

Sam Pollard
github.com/sampollard/learn/learnR
April 18, 2016

## 1 Introduction

This guide is meant for people with little or no programming experience. That being said, this guide moves quickly and is aimed towards people with technical training in some scientific field. There are many guides for programming which are aimed at computer scientists and use lots of jargon computer scientists have heard for years. This can be difficult to understand. On the other hand, many guides use such simple examples that you can't really do anything interesting with what you learn. This guide attempts to provide meaningful examples and explain the fundamental programming concepts which cannot be easily Googled. This guide comes with a supplementary R file which contains all of the example code used in this guide. If you have any feedback please mail me at `sam.d.pollard@gmail.com`

A word on notation: things written in `this font` represent things which are typed exactly into R or the output of running commands. Things in ***this font*** are vocabulary terms, and things in *this font* are mathematical notation or emphases added by me. It should be clear from the context.

R is what is called an ***open*** project. That means that anyone can contribute to R. This results in a large number of what are called ***packages***, which consist of a bunch of R files which "work together" to provide a service. This also results in R being really really huge in terms of the number of features it provides.

To use R, you must first download and install it. I found the easiest way to do this is to go to `http://cran.cs.wwu.edu/`, which is hosted by Western Washington University. You then install it like any normal program.

There are two basic ways to work with R. One is in ***interactive*** mode, the other is ***batch processing***. Batch processing is when R executes commands it reads from a text file. This text file is called ***source code*** and by convention these files end with the `.R` extension. This can be done in Windows by clicking *File* then *Source R Code...*. Alternatively, you can edit an R source file by clicking *File* then *Open Script*. Interactive mode is useful for temporary or scratch work, while batch processing is useful for reproducibility.

For example, you can run the file `sample_code.R` provided via batch processing. This will run every command presented in this guide and save a few files in the same folder (a.k.a. directory) as the one where the source code is stored. This is far from being pedagogical because everything will finish in less than a second, but this behavior is critical for automating tasks.

When you start R the default behavior is to run in interactive mode. Below are a few sample commands in interactive mode. When R first starts in this manner it will display some text about what R is and how to exit. This has been omitted. As this guide progresses I

recommend you execute each of the commands from the attached file `learnR.R`. Don't worry if much of this doesn't make sense; each component will be explained.

```
1  > data(trees) # Load a sample dataset
2  > nrow(trees) # Count the sample size
3  [1] 31
4  > colnames(trees)
5  [1] "Girth"  "Height" "Volume"
6  > max(trees["Volume"]) # Find the largest in the column
7  [1] 77
8  > max(trees[3]) # The same as above because we access the third column
9  [1] 77
10 > apply(trees, 2, mean)
11    Girth   Height   Volume
12 13.24839 76.00000 30.17097
```

The symbol `>` is called a ***prompt***, which is the interpreter's (we'll define what that is later) way of saying, "I'm ready to process another command." Anything following a `#` is a ***comment***. This is to make the commands more human-readable, but are ignored by R entirely.

Line 1 loads in a sample dataset named `trees` with which we can experiment. To see all the available sample datasets, type `data()`.

The `[1]` symbol on Line 3 is a way of saying, "here is the first element of the output which you're not storing anywhere."[1] As we will see, it is common to store the output (or ***return value***) of a function call in a variable. This is explained in the next section, but in this example the return value of the function calls are just printed[2] for the user to view.

Next up, we have something like `nrow(trees)` which takes as input the variable `trees` and returns the number of rows that variable contains. Typing a function in R is the same as in math: the function name followed by the arguments (or inputs) in parentheses.

The last command is a bit complicated but fits with intuition. The `apply` function takes as an argument some ***data frame*** (in this case `trees`), a 1 or 2 (1 for rows, 2 for columns), and a function (`mean` here), and applies that function to each row or column in the data set. It is generally considered good practice to use built-in functions such as `apply` because they make code easier to understand and maintain.

It is important to keep in mind while programming R that almost everything is a function call. Functions in R are much more general and abstract than what the average person (read: non-mathematician) has learned about functions. In math we write $f(x, y) = x^2 + y^2$ to describe a function $f$ which takes in two numbers and returns a third number. Functions in R don't need to just take as input numbers. For example, `trees` is not a number but rather a table of numbers. Functions are so important in R that after reading this guide the word "function" might sound weird. In fact, functions are so important that they deserve their own section (Section 6).

[1] Technically, it refers to the first element of a one-element vector. These will be explained later.

[2] It is common when programming to use "print" for describing something displayed on the screen instead of just something physically printed.

You may also see a `+` as a prompt which means essentially, "you're not done yet." For example, if you were to type `5 *` and then press enter, you will see (comments added by me)

```
> 5 *         # 5 times what?
+ 5           # The * is multiplication in R.
[1] 25
>
```

This can also occur because of unclosed parentheses or braces. If you are not sure why this is occurring, you can type the Ctrl and C keys simultaneously in Linux to cancel whatever it is you were typing and get back to `>`. This may also be denoted as Ctrl+C or even `^C`. The Escape key accomplishes this task in Windows.

# 2 Basic Syntax and Examples

R is what is called an ***interpreted*** language. This means that there is an aptly-named ***interpreter*** which keeps track of all the variables you have and how to execute the commands you type. The benefit to being interpreted is you have the flexibility of running one line at a time and thus can easily run in interactive mode.

One potential drawback of an interpreted language is it can be difficult to find the errors in your code, because they may not be detected until the source code is run. This can be mitigated by testing small chunks of your code as you write it.

In the first example, the object `trees` is what is called a ***data frame***. In R, this is denoted `data.frame`. This can be thought of as your typical spreadsheet format: A list of a bunch of columns. The one restriction is that each column must have equal length. Here is a quick example:

```
1 > name <- c("H","He","Li")        # c can be thought of as "combine"
2 > mass <- c(1.0079,4.0026,6.941)  # i.e. make a vector from the arguments
3 > atomic_number <- seq(1,length(name))
4 > mydf <- data.frame(atomic_number, name, mass)
5 > mydf                           # Print out the data.frame
6   atomic_number name   mass
7 1             1    H 1.0079
8 2             2   He 4.0026
9 3             3   Li 6.9410
```

A few remarks are in order. The most fundamental is the effect of `<-`. This performs an ***assignment*** of the right-hand value to the left-hand value. The assignment on line 1 creates a variable called `name` and puts a value inside of it. The `<-` symbol is read as "gets." This is the official way to perform assignment, but you will often see code such as `name=c("H","He","Li")` instead, which is also assignment (this is common syntax in other programming languages). I recommend using `<-` because `=` is used in another context in R so this clarifies the distinction. The fourth line calls the `data.frame` function which takes as input some number of vectors and returns a data frame. This return value is stored in

the variable `mydf`. The name, mass, and atomic_number form the ***header*** (also known as column names) of the data frame.

In Line 3, we generate a *sequence* using the `seq` function. The first argument is the starting value, and the second is the ending value.

Unlike most other programming languages, the period can be used in variable names. It is typically used to denote a more specific way to go about things. For example, the function `write` is generally used to write an R object to a file, but functions like `write.csv` or `write.table` specify how the output file looks. Now suppose we want to share our work with others and wish to save `mydf` as a file.

```
> write.csv(mydf, "report.csv")
```

The `write.csv` function takes in at least 1 argument: the file name. To prevent R from interpreting `report.csv` as a variable, the value is closed in quotes. A bunch of letters enclosed within quotes is called a ***string***[3].

Here is the resulting file `report.csv`:

```
"","atomic_number","name","mass"
"1",1,"H",1.0079
"2",2,"He",4.0026
"3",3,"Li",6.941
```

The first column was filled in for us. By default, `write.csv` writes the first column of the csv as a bunch of row names. This may be useful in some cases, but for us it just adds a duplicate of the second column. Now, we can read this into a new data frame in a similar fashion:

```
> read.csv("report.csv")
  X atomic_number name   mass
1 1             1    H 1.0079
2 2             2   He 4.0026
3 3             3   Li 6.9410
```

But that doesn't look quite right. Now, us typing `write.csv(mydf, "report.csv")` is the simplest way we can write to a file. However, it makes things more difficult for the person reading your file. To make the input simple to read, we need to use ...

## 2.1 Optional Arguments

If you were to type `?read.csv` into R there would be a lot of parameters specified under *Usage*. Arguments after the required arguments are called ***optional arguments*** or ***optional parameters***. Parameter specification takes the general form `tag = value`, where `tag` is specific to the function being called and `value` can be anything you want, though not every value is correct in every circumstance. This appears all over the place in R programs, and here we don't have the choice to use `<-`. For example, since we didn't specify a `sep` in `read.csv`,

[3]Here are other examples of strings: `"Hello"`, `"121a.a;lk!"`.

`sep` takes the value of `","` by default. Keep in mind that the ***tag*** of optional arguments are *not* variables outside of the function. These options only make sense inside of the function in which they are being called.

Going back to our `write.csv` example, we wish to not include the row names because they complicated things for `read.csv`. This may be done simply as

```
> write.csv(mydf, "report.csv", row.names = FALSE)
```

Done this way, we get

```
> read.csv("report.csv")
  atomic_number name   mass
1             1    H 1.0079
2             2   He 4.0026
3             3   Li 6.9410
```

which is exactly what `mydf` looks like.

# 3 Some Subtleties

## 3.1 Vectors

The previous example used the `c` function. This is one way to gather data in the correct format. This is an important part of R and programming in general. The distinction here is between a ***vector*** and a ***scalar***. This fits with mathematical intuition but not so much with the real world. Ask a mathematician what a vector is and she will respond, "It is something that behaves like a vector." While this is sort of a joke, there really isn't any better definition. In R, vectors are all over the place. Here are some examples and a non-example (can you spot which?)

```
n <- nrow(trees)           # The number of trees
tag <- seq(1,n)            # A sequence from 1 to n, inclusive
lifespan <- 70*runif(n)    # n uniformly distributed random numbers in [0,70]
volume <- trees["Volume"]  # The volume of each tree
```

The last example is *not* a vector. It is a data frame. This means things that require vectors won't work on the data frame. For example,

```
> mean(volume)
[1] NA
```

However, if we make a small modification:

```
> volumevec <- trees[["Volume"]]
```

This *is* a vector.

If you're stuck, the `class` function may describe what sort of data you're dealing with. Another useful method is `ls()` which lists all the variables in your current workspace.

But what about the first example? Is it a vector? In some sense, yes. A vector can have one element. It is easiest to think of a scalar as the special vector which has one element. In general, most things you can do with scalars you can do with vectors. For example,

```
> lifespan <- lifespan + 15
```

would increase the `lifespan` variable by 15 (add 15 to each element in lifespan). Notice that this statement contains lifespan on the right-hand side and the left-hand side. In English we would say, "take each element in lifespan and add 15 to it, storing the result in the variable lifespan."

## 3.2 Building Up

A common task when working with spreadsheets is combining data from multiple sources into a single structure. Here, we will look at the `cbind`, and `paste` functions. We will keep using our previous examples and create a data frame which represents all of the vectors we have created. We begin this somewhat contrived example by creating "names" for each of the trees.

```
> treenames <- paste("T", tag, sep = "")
```

This creates a sequence of strings. Notice that R is smart enough to determine that while you only specified a single string "T" (instead of a vector of strings of the same length as `tag`), it concatenates "T" with each element of `tag`. For example, tree 15 would be named `T15`.

To make our tags look nice, we must specify the *separator* `sep`. This may be any string and `sep` is used in many other functions as well. We set `sep = ""`, a.k.a. the ***empty string***, a.k.a. the string with no characters. If this was left out, i.e. `paste("T",tag)`, then by default the strings are concatenated using a space as separator. We would get `T 15` instead of `T15`. Why is this? The motivation behind this is that most of the time a user will want the values separated by spaces. Without anything, there is the ***default argument*** for `sep`, which is a single space `" "`.

Now, to create a new data frame from the existing data we created, try

```
> mydf <- cbind(treenames, trees, lifespan)
```

This is close to what we want. The function `cbind` takes as input data frames or vectors, and combines them by column into a new data frame. There is an analogous function called `rbind` which combines the data by row. Here is what the first few entries look like:

```
> head(mydf)
  treenames Girth Height Volume lifespan
1        T1   8.3     70   10.3 15.18891
2        T2   8.6     65   10.3 15.85794
```

```
3        T3   8.8     63   10.2 15.41223
4        T4  10.5     72   16.4 15.30183
5        T5  10.7     81   18.8 15.86208
6        T6  10.8     83   19.7 15.87640
```

By the way, `head` is a nice way to just see the beginning of a data frame (it is common to deal with thousands of rows of data). But this doesn't look quite right: the row names are just 1, 2, 3, ... but we want them to represent our cleverly named tree names. R allows us to change row names and column names using the aptly named functions `rownames` and `colnames`. So,

```
> rownames(mydf) <- treenames
```

We can also retrieve the row or column names by putting this function call on the right-hand side of an assignment like so: `names <- rownames(mydf)`. To anyone with programming experience, this is a bit strange. The function can be used both as an **l-*value***, a.k.a. the left-hand side of an assignment statement a.k.a. the location of the variable getting changed, and an **r-*value***, a.k.a. the right-hand side of an assignment statement a.k.a. the return value of a function call. This is where the `<-` notation keeps us sane: it directs us where the data we computed is being stored.

However, this data frame still looks a bit off:

```
> head(mydf)
   treenames Girth Height Volume lifespan
T1        T1   8.3     70   10.3 15.18891
T2        T2   8.6     65   10.3 15.85794
T3        T3   8.8     63   10.2 15.41223
T4        T4  10.5     72   16.4 15.30183
T5        T5  10.7     81   18.8 15.86208
T6        T6  10.8     83   19.7 15.87640
```

Let's delete that first row:

```
> mydf <- mydf[,-1]
```

That is some cryptic R code. There are other ways to accomplish the same task, but this one easily generalizes. In general, we access things by `[row, column]`. So by default, we are accessing *every* row. That is, there is nothing preceding the comma. Now, we are accessing every column but the first one (think of `-` as removing the first column). Alternatively, we could *include* every other column. So

```
> mydf <- mydf[,c(2,3,4,5)]
```

would accomplish the same task. If we only cared about a few rows of `mydf` we could write

```
> sample <- mydf[seq(4,9),]
```

This takes the 4th through 9th rows of `mydf`, inclusive.

# 4 Regression and Plotting

This section will first gather some data and curves to be plot using R's regression features. These will be plotted but the output will be the default, plain format. Next, more advanced plotting features will be exploited to improve the quality of the figures and allow saving of the figures in multiple file formats.

## 4.1 Regression

Below is an extended example which plots a bunch of (simulated) noisy data and fits a curve to it. Which realistically is the solution to about half of all scientific problems.

```
1 lb <- 0     # The lower and upper bounds will be used repeatedly
2 ub <- 10
3 time <- seq(from = lb, to = ub, length.out = 100)
4 noisy_data <- rnorm(100, mean = 0, sd = 5) + 10 # Equivalently: rnorm(100,10,5)
5 noisy_data <- abs(noisy_data) + time^2
6 # Make a scatterplot of the data
7 plot(time, noisy_data)
```

The first two lines create some made up data. Specifically, the "noise" is simulated with a normal distribution with mean 0 and standard deviation 5. Taking the absolute value of the data (`abs` on line 5) will be nice a bit later when we want to interpret and plot the data. Since the 10 is added in line 4, this puts the mean at 15. Line 5 adds this noisy data to the square of a sequence of 100 elements going from 0 to 10. For example, `time[1]^2 = 0`, `time[2]^2 = 0.0102`, ..., `time[99]^2 = 97.99`, and `time[100]^2 = 100` . We have seen `seq` before and this is a more general use of the function. Squaring the points created from `seq` gives points from the quadratic $f(x) = x^2$. Putting this all together means the noisy data *should* be best fit by the curve $x^2 + 10$. Hopefully, a least-squares approximation will yield a curve close to that.

However, since we know from what distribution the data is sampled, we are dealing in some sense with a "solved" problem. Suppose for a while that we don't know the underlying distribution of the data. A valid technique would be to guess the data fits some linear model.

```
1 linearmodel <- lm(noisy_data ~ time)
2 # Get the coefficients to overlay the best-fit line over the scatterplot
3 intercept <- coef(linearmodel)[1]
4 slope <- coef(linearmodel)[2]
5 curve(slope * x + intercept, add = TRUE)
```

Line 1 attempts to fit a linear model to the data. The `~` character (tilde) indicates a *formula* is used. With a linear model, the parameters are implied. That is, `lm` reads in the simple expression `noisy_data ~ time` and from it knows both the the data on which a regression should be performed and the approximate form of the equation. `lm` is attempting to find the best $m$ and $b$ which satisfy the equation $f(t) = mt + b$ where $f(t)$ is the noisy

data and $t$ is time. In this context, $m$ and $b$ are the ***coefficients***. The optional parameter `add = TRUE` states that the curve should be added to an existing plot. The default is false, which is to say each call to `curve` creates a new plot.

Let's see how a linear model fits:

```
> summary(linearmodel)

Call:
lm(formula = noisy_data ~ time)

Residuals:
    Min      1Q  Median      3Q     Max
-16.658  -6.204  -1.393   6.674  21.740

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -6.3943     1.8774  -3.406 0.000957 ***
time         10.0070     0.3244  30.852  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.458 on 98 degrees of freedom
Multiple R-squared:  0.9067,        Adjusted R-squared:  0.9057
F-statistic: 951.8 on 1 and 98 DF,  p-value: < 2.2e-16
```

This summary is a Type I analysis of variance table. From this, we get a statistically significant result (i.e. publishable!), however viewing the plot shows the model is lacking (Fig. 1).

To see this in more depth, we may call `plot(linearmodel)` which produces several plots of the data. This further indicates the model is biased at the endpoints, suggesting a linear model isn't ideal.

Forget for a moment that we know the model follows a quadratic function. Looking at the data one can see the plot could also be considered exponential since the growth rate increases as time increases. With a linear model, we did not assume much about the data; we just assumed a line would fit the data and let R do the work to find the best $m$ and $b$. We can do the same thing for an exponential model, though we have to be a little more clever. In general, we can describe exponential growth functions using the general form

$$f(t) = ce^{kt}$$

where $c$ and $k$ are real numbers. The $c$ is known as the starting population and $k$ is the ***growth rate***. Alternatively, since $c$ and $k$ can be any real number, let us make a new constant $b = \ln(c)$. Then we may rewrite this family of functions as

$$f(t) = ce^{kt} = e^{\ln(c)}e^{kt} = e^b e^{kt} = e^{b+kt}.$$

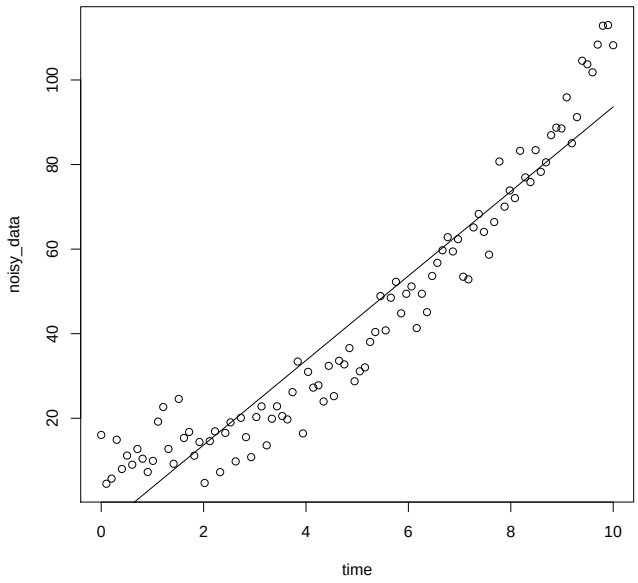


Figure 1: Our first shot at fitting some data.

This is definitely not a linear model, but suppose we take the natural logarithm of both sides and exploit some properties of logarithms. This yields

$$\begin{aligned}\ln(f(t)) &= \ln(e^{b+kt}) \\ &= b + kt,\end{aligned}$$

which *is* a linear model in the logarithm of $f(t)$. We forced all values to be positive near the beginning of the section, which is required for the natural logarithm to make sense. Fortunately, the natural logarithm is so common that in R the command to compute it is simple: `log`. Thus, instead of forming a linear model using just `noisy_data ~ time`, we instead want `log(noisy_data) ~ time`. To create this model we use

```
expmodel <- lm(log(noisy_data) ~ time)
expintercept <- coef(expmodel)[2]
expslope <- coef(expmodel)[1]
curve(exp(expintercept + expslope*x), add = TRUE)
```

Now that we have two models, how do we determine which is better? Let's look at `summary` again:

```
> summary(expmodel)

Call:
lm(formula = log(noisy_data) ~ time)
```

```
Residuals:
     Min       1Q   Median       3Q      Max
-2.51876 -0.08660  0.02944  0.13864  0.67754

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.12047    0.06883   30.81   <2e-16 ***
time         0.27124    0.01189   22.81   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3467 on 98 degrees of freedom
Multiple R-squared:  0.8415,        Adjusted R-squared:  0.8399
F-statistic: 520.2 on 1 and 98 DF,  p-value: < 2.2e-16
```

One way to judge which model is better is to look at the residual standard error (RSE) of both models. The linear model has an RSE of 9.458 while the exponential model has just 0.3467. Furthermore, overlaying the two models indicates the exponential model seems to look better (see Fig. 2). However, visualizing the data can be difficult for data in higher dimensions.

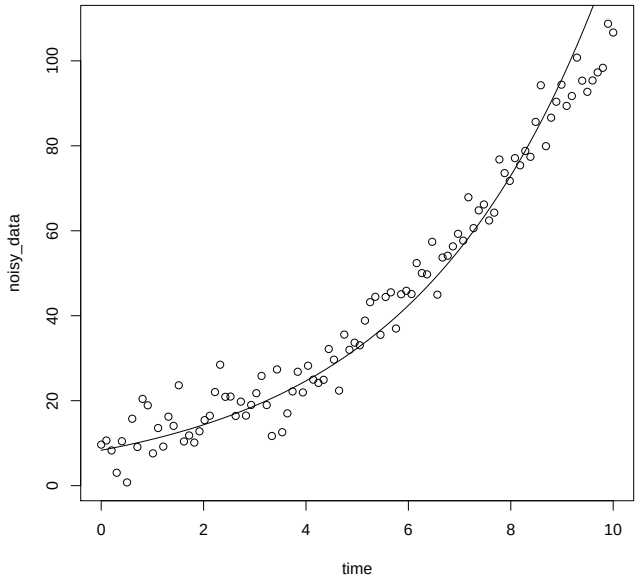


Figure 2: Creating an exponential model.

However, we know the answer is quadratic because it was made that way. To fit a nonlinear curve to the data we may use a different function, `nls` (nonlinear least squares). Here is how that would be performed:

```
1 quadratic <- noisy_data ~ p1*time^2 + p2
2 model <- nls(quadratic, start = list(p1=0, p2=0))
```

```
3 # Plot the new, least-squares quadratic
4 p1 <- coef(model)[1]
5 p2 <- coef(model)[2]
6 curve(p1*x^2 + p2, from = lb, to = ub, add = TRUE)
```

Line 1 is creating a general formula. This time, we have to specifically state the parameters `p1` and `p2`. Notice that these are not variables we have declared. That's okay; they are simply placeholders telling R what parameters it can mess with to get the best curve to fit the data. In Line 2, we specify using the optional parameter `start` that `p1` and `p2` should both start at zero. This is not required, but if you happen to know about how the data should look, rough estimates of the parameters can help `nls` arrive at a good solution. Here, using the quadratic model fits even better because that was exactly how the data was created.

## 4.2 Plotting

R has very powerful plotting features. They are described as "beautiful" by data visualization geeks. One reason for this is that R can output in vector formats, which is to say they scale to any size. R supports many filetypes, but this guide uses the pdf format by calling the `pdf` function. Two more useful filetypes supported are png and jpg, which are called identically to the pdf driver, only using the `jpeg` and `png` functions.

This section will explain the various uses of the `plot` and `curve` functions used in the previous subsection and expand upon them to produce more sophisticated graphs.

Now, suppose we have already created all the required data and gone to assemble a plot like so: (this is simply a compilation of previously shown commands)

```
plot(time, noisy_data)
curve(slope * x + intercept, add = TRUE)
curve(p1*x^2 + p2, add = TRUE)
```

Below is one way to annotate the plot and save it to a pdf.

```
1  pdf("sample_plot.pdf")
2
3  # Make the scatterplot and label the axes
4  plot(time, noisy_data, xlab = "Time (months)",
5       ylab = "Number of Cats (millions)")
6  # Change the title and marginal text
7  title("Number of Cats Over Time", line = 2, cex.main = 1.6)
8  mtext("What do we do with all these cats?\nBy Sam Pollard", font = 3,
9        cex = 0.8)
10 # Overlay the linear and quadratic best fit curves
11 curve(slope * x + intercept, add = TRUE, lwd = 2, col = "red")
12 curve(p1*x^2 + p2, add = TRUE, lwd = 2, col = "blue")
13 # Create the legend
14 quadtext <- paste0(format(round(p1, 2), nsmall = 2), " t^2 + ",
15                    format(round(p2, 2), nsmall = 2))
```

```
16 linetext <- paste0(format(round(slope, 2), nsmall = 2), " t + ",
17                    format(round(intercept, 2), nsmall = 2))
18 legendtext <- c(quadtext, linetext)
19 legend("topleft", legend = legendtext, col = c("blue","red"), lwd = c(2,2))
20 # Say that we're finished plotting so the pdf can be saved
21 dev.off()
```

This is a lot of code to take in at once but it will be broken down. The first call to plot on line 4 plots time against the data. The x and y labels are changed from their default (the variable names) to something sillier. For line 6, the `line` parameter sets where the title should appear, as determined from an offset from the top of the graph. That is, `line = 2` is two "units" above the top of the graph, while `line = -2` would be two units down from the top, so inside of the graph. Exactly what determines a unit is up to R to decide. In general, it is a good idea to trust R in what looks nice. One must give up a little freedom for convenience, knowing that things are done in R for good reason. The second parameter `cex.main` stands for "character expansion of the main text." This means that, relative to the default size 1, the title is 1.6 times larger than that.

Line 8 creates "marginal text" and the parameters are a bit cryptic. The `\n` in the middle of my string is a newline, so my name appears under the "Cats" query. Setting `font = 3` means italic, and `cex = 0.8` shrinks the text a bit.

Lines 11 and 12 overlay curves on the scatterplot. The first argument is a general form for a function: notice that `x` is not a variable which has a value associated with it. This allows functions to be plot. The `lwd` parameter stands for "line width" and is again a multiple of the default value 1.

Next, the legend is created. This is done in several steps so the R command doesn't get too messy. The `quadtext` and `linetext` store the string which represents the equation. There are many ways to do this, and R does support mathematical equations. However, I used the `paste0` function. Recall that `paste` allows concatenation of strings. However, in the previous usage, we created a vector. Next, `paste0` is simply a convenience and is identical to `paste(..., sep="")`. That is, this function combines everything you give it into a single string. The `round` function is as one would expect: rounding a value to 2 decimal places. The second argument to `format` gives the minimum number of digits to the right of the decimal place.

To create the legend, one must first specify the location. I chose the top left corner, but there are also options such as `top`, `bottomright`, or by the x and y coordinates. Notice that all the parameters afterwards are vectors created using the ubiquitous `c` function. Thus, the first element of each parameter will correspond to the first line and text to be displayed with that line.

Lastly, once you're satisfied with the graph the function `dev.off()` turns off the device which is displaying the plot which causes the file to be written to the location specified by `pdf`. Running this script will cause a file called `sample_plot.pdf` to be saved in the ***working directory*** that R is running in. To specify exactly where the plot should be saved it is best to use an absolute file location. For example:

```
pdf("C:\Users\Sam Pollard\Documents\R\sample_plot.pdf")
```

All this combined makes the graphic shown in Fig. 3

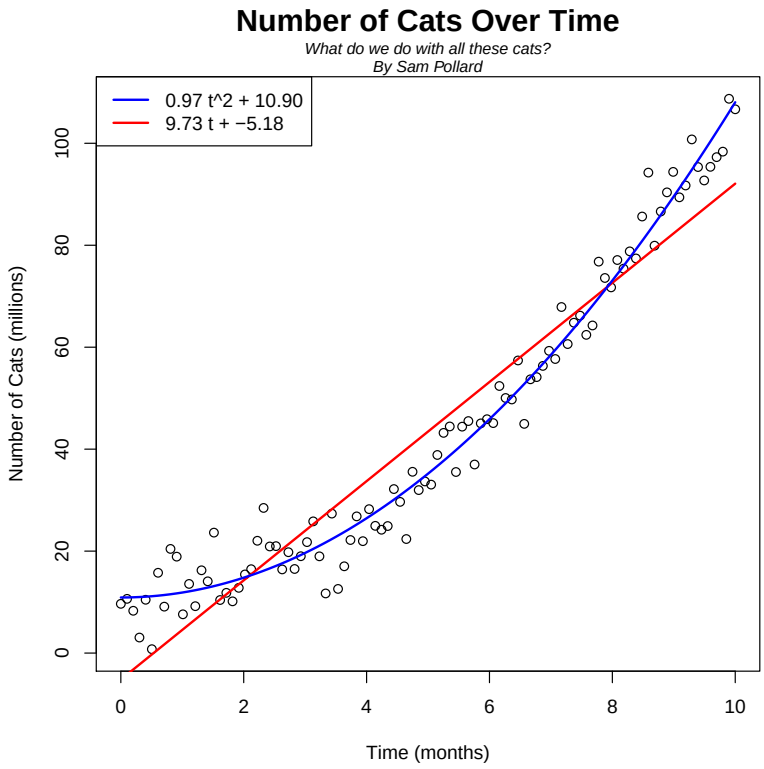


Figure 3: The figure created in Section 4.2.

# 5 Conclusion of Part I

This concludes the first part of the guide. You are missing many important aspects of programming in general. Much of what makes a programming language a programming language is missing: the `if` and `for` statements, creating custom functions using `function`, and much more. This is firstly a practical guide and even without those fundamental features a lot can be done with R.

# Part II
# Learn You Some R, "In Theory"

This part of the guide attempts to give some general programming knowledge as opposed to some useful tricks. This can be thought of as learning how to build a fishing rod instead of learning how to fish. The benefit to the theoretical side is these concepts apply to every other programming language. The drawback is even if you know how to build a fishing rod, you can still be a terrible fisher.

## 6 Functions

It is difficult to give a meaningful definition of a function, but the best I can come up with is, "a function is an object which takes as input zero or more arguments, performs some operations based on these arguments, and then returns a value or nothing." So, what is a function?

Incredibly useful, as it turns out. Without getting into too much detail, it has been proven that *everything* computers can be made to do can be boiled down to computing functions and setting variables[4]. This is motivation to understand both variables and functions.

Now, let's start by creating our own function in R. Suppose we just want to take any vector as input, and enclose that in parentheses.

```
parenthesize <- function(x) {
        interior <- paste(x, collapse = " ")
        return(paste("(", interior, ")"))
}
```

Notice that this looks like variable assignment. In fact, this is exactly what is happening. Next up we see the braces `{}`. All these do is define what is inside the function. Programmers usually write functions which take up multiple lines of source code, so the braces show exactly when the function begins and ends. It is proper style to indent everything between sets of braces. This makes the code more readable because there may often be multiple levels of braces which can get incredibly confusing even with proper indentation.

Now the function `parenthesize` is ***bound*** to the variable `parenthesize` and we can use it like we would use any other built in function:

```
> parenthesize(c(1,2,3,4))
[1] "( 1 2 3 4 )"
```

In R, `return` is a special function which exits out of the current function and passes its argument to whatever called the current function. In the example above, the ***caller*** is the R interpreter. The R interpreter gets the return value of `parenthesize(c(1,2,3,4))` and then since it has no variable to store it in, prints the value to the screen. We could have bound a return value of `parenthesize` to a variable like we have done with other functions.

[4] If you would like to learn more, this is referred to as Lambda Calculus.

The variable `interior` in this context has what's called a ***scope***. This is an important concept in programming and refers to when you can access particular variables. For example, once we define the function, if we try to access `interior` outside of the braces defining the function we get an error:

```
> interior
Error: object 'interior' not found
```

This is because `interior` is local to the function `parenthesize`. This helps keep the set of all active variables less cluttered, and makes things easier because you can reuse variable names such as `x`. On the topic of `x`, by making it an argument to `parenthesize`, we are making the variable `x` only in scope between the braces following the key word `function`. That is, whatever the value of `x` gets inside of the function is forgotten when the function returns. This is useful because we can now pass anything into `parenthesize` and it will always be known as `x` inside of the function. Even if we had a variable called `x` that we were using *outside* of the function definition, this would be temporarily ***shadowed***, or eclipsed by the `x` specified as `parenthesize`'s input argument.

## 6.1 A (Seemingly) More Complex Function

Again, we address the problem of correctly formatting data. We have seen that `read.csv` attempts to make this easier with its `sep` argument. This allows us to deal with an arbitrary separator for a csv file. However, many other applications are not as forgiving. For example, a URL must be in a very specific format to get to the correct page. As a motivating example, suppose we have the following protein IDs (as recognized by the RCSB Protein Data Bank) and wish to convert these into a comma-separated list:

```
1 > # The input
2 > proteins <- c("3G73","3CO6","1VDE","1AF5","1EVX","2OST","1QQC","1KN9","3BF0")
3 > # The desired output
4 > protein_string <- "3G73,3CO6,1VDE,1AF5,1EVX,2OST,1QQC,1KN9,3BF0"
```

The reader may notice the second line above is basically already a comma-separated list. Suspend your disbelief and suppose that we didn't set `proteins` by hand, or that you don't want to remove all of those quotes around each protein ID. Our goal then is to have some function `intersperse` which takes as input a vector and outputs that vector with commas in between the entries. So instead of typing the value of `protein_string` out by hand like I did when writing this guide, we could get the same result with a function.

### 6.1.1 A Pedagogical but Nasty Example

Full disclosure: there is a much simpler way to do this using functions we have already seen before. These will be shown later in this section. However, for the sake of generalization I will explain this in terms of two fundamental programming language constructs: the `for` loop and the `if` statement. For starters, only look at the first 7 lines.

```
1  intersperse <- function(vec, ele) {
2          vlen <- length(vec)
3          if (vlen >= 1) {
4                  y <- vec[1]
5          } else {
6                  return("")
7          }
8          if (vlen > 1) {
9                  for (i in seq(2,vlen)) {
10                         y <- paste(y, ele, vec[i], sep = "")
11                 }
12         }
13         return(y)
14 }
```

We begin `intersperse` by getting the length of our input vector. Next, we check the length of our input vector using an `if` construct. The general form of this type of statement is

```
if (this condition is true) { # then
        Do everything here
} else { # Optional and executes if the condition in parentheses is false
        Do everything here
}
```

Let's break this down further. Up until the mid-1800s, if you were to show a mathematician something like $3 = 4$ and ask, "how do you interpret this?" he would probably just say "that doesn't make sense, you idiot." But after the pioneering work of George Boole, we are now able to see $3 = 4$ and give it meaning, specifically, "false." Why is this useful? Because when we see something like `vlen >= 1`[5] in Line 3, we are not *declaring* that `vlen >= 1`, we are *asking* if `vlen >= 1`. It may not be.

Thus, in Line 3, if `vlen` is in fact less than one Line 6 is executed and the function returns the empty string. This means nothing else in the function is executed. This was exactly my desire: I didn't want the function to return `y` on line 13 if there were no elements in `vec` with which it could work.

At a high level, there are three cases to consider when building our `intersperse`:

1. The vector has no elements in it (Line 6)
2. The vector has just one element (Line 4 executes but Lines 9–11 don't)
3. The vector has multiple elements (Lines 4 and 9–11 execute)

The reason to split cases two and three is we desire

[5] By the way, `>=` is the R way to write $\geq$

```
> intersperse(c(1,2), ",")
[1] "1" "," "2"
```

instead of

```
> intersperse(c(1,2), ",") # WRONG
[1] "1" "," "2" ","
```

The algorithms which arise from making this sort of distinction are called ***fencepost algorithms***. The analogy is such: when building a fence, you need to start with a fencepost and then add on the wire followed by another post until completion. Thus, we start with our "post" (Line 4) and then add the "fence" (`ele`) followed by another post (another element of `vec`).

If `vlen > 1` is true, there is some extra work to do and for this we need the `for` construct. Words like `if` and `for` are called ***reserved words***. This means we cannot use them as variables; they are too special. Line 9 shows why `for` is special: it allows repeating the commands contained in its proceeding braces. The programming jargon for this repetition is a ***loop***. The statement `for(i in seq(2,vlen))` translated to English is, "assign `i` to each element of the vector returned by `seq(2,vlen)` in order, then do the following commands." The code between the braces is called the ***loop body***. Suppose `vlen` were 4. Then the loop body would execute the following commands:

```
y <- paste(y, ele, vec[2], sep = "")
y <- paste(y, ele, vec[3], sep = "")
y <- paste(y, ele, vec[4], sep = "")
```

Now, we don't have to guess or know ahead of time how large `vec` is. We can just get its length and let the computer do the work.

### 6.1.2 A Simpler Version

Notice that at each iteration of the loop, we overwrite the variable `y`. This is actually rather wasteful, because we have to create a whole new string for each iteration. While this may not make a noticeable difference for small sizes of `vec`, when programming we must be aware of the runtime of each line of code we write. For example, if we are doing work on large datasets and are constantly overwriting large variables, this can make our code run very slowly.

One solution to this is to ***preallocate*** space. Every variable takes up a certain amount of memory. If the variable needs to be expanded in R usually R has to create a new place with sufficient space and copy all of the data from the old location to the new location[6]. The exception of this is with scalars like numbers. The way numbers are stored in R allows pretty much any value you would want to work with to be stored in a constant amount of space. In particular, 64 bytes. For example, like `x <- x + 2.5` will not need to allocate extra space in the case that `x` gets too big[7].

One way to think of this is when you first create a variable like

[6]Note that this is not true for all types of data in R. Some are able to grow without copying all of the old data, for example the `list` data type.

[7]R uses ***double precision floating point*** numbers for all of its numerical data.

```
> y <- "Once upon a time"
```

this can be thought of as buying a notebook which can only hold space for 16 characters. If you do something like

```
> y <- paste(y, "there was a very exciting R guide online.")
```

your "notebook" is already full, so you must buy a new notebook which can hold space for $16 + 1 + 41 = 58$ characters and copy everything into that notebook. This sounds rather silly when talking about notebooks but this is essentially how R handles variable assignment.

One last thing to keep in mind is *why* you are programming in R in the first place. Oftentimes it will be because you don't want to do something by hand, or doing something by hand would take a prohibitively long time. Likewise, the business of copying your notes into new notebooks sounds awful by hand, but computers are fast at doing these things. When dealing with such small sets of data, the time difference is between maybe 0.001 second and 0.01 second. Big deal. You may hear this also as an argument as to why R is not a "good" programming language; it is slow. Of course, R is not meant to be used in every context. There are many things in R that are incredibly easy compared to other languages. The power of R is not in how quickly the computer can run your code, but rather how quickly you as a programmer can accomplish your goal. With this in mind, one important philosophy of programming is to never ***preoptimize*** your code. Yes, the first version of `intersperse` I wrote may not be the best version possible. However, if it works for your purposes, that is good enough.

Well. I kind of went off on a tangent there. The whole point of this is to *simplify* the `intersperse` function. Without further ado, here is the next iteration:

```
1 intersperse <- function(vec, ele) {
2         if (length(vec) < 2) {
3                 return(vec)
4         }
5         y <- vector("character", length = 2*length(vec) - 1)
6         y[seq(2, length(y), 2)] <- ele
7         y[seq(1, length(y), 2)] <- vec
8         return(paste0(y, collapse = ""))
9 }
```

Line 5 is the part which preallocates a vector to store the return value. We specifically tell R how much data to preallocate: twice the length of the input vector minus one. The last line of the function, Line 8, uses the `paste0` function with the `collapse` option. This is to collapse our vector into a string. As before, this is a way to format the data correctly so we get the correct output. To see why this is required, try changing the last line to just `return(y)`. What sort of output do you see?

### 6.1.3 KISS (Keep It Simple, Stupid)

We have seen the `paste` function before as well as the `collapse` option. Now, what if we just let R handle the hard work for us?

```
intersperse <- function(vec, ele) {
        return(paste(vec, collapse = ele))
}
```

To ensure `intersperse` gives the correct output, we have to bring out the first ***test case*** of the guide.

```
> protein_string_with_intersperse <- intersperse(proteins, ",")
> protein_string_with_intersperse == protein_string
[1] TRUE
```

This is good practice: for each little chunk of code you write, it is good to write tests to make sure the code is doing what you expect. We can further automate this by using the `stopifnot` funcion. This will return an error if any of the arguments you pass it evaluate to `FALSE`. This way, we don't have to look at results and make sure they are all true, we can just insert these into our R source code. Here are a few more test cases:

```
stopifnot(intersperse(proteins, ",") == protein_string,
          intersperse(c("Just one"), "-") == "Just one",
          intersperse(c(), "") == "")
```

For this method of testing, no news is good news. That is, nothing is said if the program is functioning correctly. This is nice to avoid too much output when running your programs.

# Part III
# Exercises

Just as you cannot learn how to play piano by watching someone else play, you cannot learn to program by watching—or rather, reading—someone else program.

1. Write your own `head` function. Call this function `Head`. Notice that this is considered an entirely different function in R. This is because R is case sensitive. Likewise, the variable `X` is different from `x`. Recall that `head` takes as input some object and outputs the first few rows (or elements) of that object. Test your function by running the following code:

   ```
   Head(seq(1000))
   data(Puromycin) # Load in a sample data.frame
   Head(Puromycin)
   Head(5)
   ```

   Make sure this matches the ouput of calling `head` with the same inputs.

2. Write a function that asks the user for his/her name and outputs the name in the form last name, first initial. For example

   ```
   > changeName()
   Type your name: Sam Pollard
   [1] Pollard, S.
   ```

   To do this you will need the `readline` function. To get more information about this function, remember you can type `?readline` into the interpreter. This will open your web browser in Windows or just display the text in the terminal in Linux. If in Linux, you can scroll up and down with the arrow keys and can quit by typing `q`.

   To further refine this, consider cases where the user accidentally presses enter without typing anything. Furthermore, what about people with last names with spaces in between such as Johannes van der Waals?

3. How would you create a quadratic model using the `lm` function? This will require transforming the data.

# 7 Resources

Arguably the most useful resource you can make use of in R is the `help` function. You pass `help` any function as an argument, which will direct you to an online help source. For example, `help(rbind)`. This is the same as `?rbind`.

Here are some other resources. I have not cited every one of the sources I used when creating this guide, but the ones omitted were almost all from stack exchange, the R documentation, or Wikipedia. Thus this can be also thought of as a partial bibliography.

[1] `http://learnxinyminutes.com/docs/r/` This is a pretty basic guide but is the easiest to follow.

[2] `https://github.com/sampollard/pcrystal` This is some of my own R source code.

[3] `http://cran.r-project.org/doc/contrib/Short-refcard.pdf` This is a useful cheat sheet. Much of it won't make sense right away, but will help you with commonly-used functions.

[4] `http://cran.r-project.org/doc/manuals/R-intro.pdf` This is a more in-depth guide to R. Notably, chapter 12 contains a lot of good information on graphics.

[5] `http://www.walkingrandomly.com/?p=5254` I used this reference in my nonlinear regression example.

[6] `http://www.statmethods.net/advgraphs/` Provides explanation of many of the parameters of the `par` and `plot` functions.