layout: default
title: Home

An intro to R for new programmers

This is an introduction to R. I promise this will be fun. Since you have never used a programming language before, or any for that matter, you won't be tainted by real programming languages like `Python` or `Java`. This is good - we can teach you the R way of doing things.

jsforcats?

Yep, this is a total rip off of JSforcats.com - hopefully Max doesn't mind.

What will we do?

- Using the R console - let's dig our claws in
- vector's - the basic R data structure
- data.frame's - weird but familiar
- lists
- indexing
- functions
- Using packages
- No no's for cats using R
- Do do's for cats - or things to do
- Open data from the web! Cat's love open data
- Reading

R console

Writing code is fun. Since you're a cat, not having opposable thumbs may be a bit of an issue, but surely you're clever enough to find a way around that.

So open up R, and you'll see something like this:

You can do math:

```
1 + 1
```

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

Type a set of letters together (also known as a *word*) within quotes and the console will print it back to you

```
"Hello Mr Tickles"
```

```
## [1] "Hello Mr Tickles"
```

Another thing you'll want to do as a cat using R is assign things to a name so that you can use it later. This is as if you were a chipmunk and you buried a nut in the ground to dig up later. You can assign anything in R to a name, then use it later (in the current R session of course :)).

Assign the number 5 to the name `mynumber`

```
mynumber <- 5
```

Later you can use `mynumber`, like adding it to another number

```
mynumber + 1
```

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

Sweet!

Vectors

Vectors are one of the simplest and common objects in R. Think of a vector like a cat's tail. Some are short. Some are long. But they are are pretty much the same width - that is, they can only contain a single data type. So a vector can only have all `numeric`, all `character`, all `factor`, etc.

But wait, how do we make a vector? The easiest way is to use a function called `c`. So `c(5,6,7)` will create a vector of numbers 5, 6, and 7. Let's try to put a `character` and a `numeric` together.

```
c("hello", 5)
```

```
## [1] "hello" "5"
```

Notice how the output of the above converted the 5 to a character type with quotes around the `5` to make it `"5"`, i.e., or an object of type *character*. But we can happily make a vector of the same type of information, like

```
c(5, 8, 200, 1, 1.5, 0.9)
```

```
## [1]   5.0   8.0 200.0   1.0   1.5   0.9
```

Vectors are handy because they can be combined to make other R objects, such as lists (see lists below), and data.frame's.

In addition, you can do something to each part of the vector. Let's say you have a vector of three types of dog:

```
dogs <- c('dalmations','retrievers','poodles')
```

You can add something to each of them like

```
paste(dogs, "are dumb")
```

```
## [1] "dalmations are dumb" "retrievers are dumb" "poodles are dumb"
```

Data.frame's

A `data.frame` is one of the most commonly used objects in R. Just think of a `data.frame` like a table, or a spreadsheet, with rows and columns and numbers, text, etc. in the cells. A very special thing about the `data.frame` in R is that it can handle multiple types of data - that is, each column can have a different type. Like in the below table the first column is of `numeric` type, the second a `factor`, and the third `character`.

```
df <- data.frame(hey=c(5,6,7),
           there=as.factor(c("a","b","c")),
           fella=c("blue","brown","green"))

df
```

```
##   hey there fella
## 1   5     a  blue
## 2   6     b brown
## 3   7     c green
```

Notice that the first *column* of numbers are actually row names, and are not part of the `data.frame` *per se*, though are part of the *metadata* for the `data.frame`.

We can quickly get a sense for the type of data in the `df` object by using the function `str`, which gives information on the types of data in each column.

```
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ hey  : num  5 6 7
##  $ there: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ fella: Factor w/ 3 levels "blue","brown",..: 1 2 3
```

Matrices

Think of a matrix in R like a `data.frame` with all the same type of data, only numeric, only character, etc. A matrix is technically a special case of a two-dimensional array.

We'll not dig into these further since I'm guessing since you're a cat, you'll be more of a `data.frame` kind of animal.

Lists

Lists are sorta crazy. They are kinda like vectors, but not. Using our cat tail analogy again, lists are like cat tails in that they can be short or long, but they can also vary in width. That is, they can hold any type of object. Whereas vectors can only hold one type of object (only `character` for example), lists can hold for example, a `data.frame` and a `numeric`, or a `data.frame` and another `list`! The way we make a list is via the function `list`

```
list(1, "a")
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
```

A nested vector

```
mylist <- list(1, list("a","b","c"))
mylist
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [[2]][[1]]
## [1] "a"
##
## [[2]][[2]]
## [1] "b"
##
## [[2]][[3]]
## [1] "c"
```

Just like vectors, you can do operations on each element of the list. However, since lists can be nested you have to worry about what level of nesting you want to manipulate.

For example, if we take the `mylist` list from above, the following

```
length(mylist[1])
```

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

```
length(mylist[2])
```

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

Gives a length of 1 for each element of the list. But wait, aren't there three things in the second slot of the list ("a","b","c")? Indeed there are

```
length(mylist[2][[1]])
```

```
## [1] 3
```

Indexing

Okay, so let's say you have made a `vector`, `list`, or `data.frame`. How do you get to the things in them? Its slightly different for each one.

There is a general way to index objects in R that can be used across `vectors`, `lists`, and `data.frame's`. That is the double square bracket: `[]`. For some objects you can index by the sequence number (e.g., `5`) of the thing you want, while with others you can do that, but also index by the character name of the thing (e.g., `kitty`).

vectors

Vectors only have one dimension, as we said above. So with `[]` there is only one number to give here. For example, let's say we have the vector

```
bb <- c(5,6,7)
```

We can index to each of those 3 numbers by the sequence of it's place in the vector. Get the 6 by doing

```
bb[2]
```

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

You can also have a named vector. What's that? A named vector is like `bb` above, but each of the three elements has a name.

```
bb <- c(5,6,7)
names(bb) <- c('hey','hello','wadup')
bb
```

```
##   hey hello wadup
##     5     6     7
```

```
names(bb)
```

```
## [1] "hey"   "hello" "wadup"
```

With a named vector we can get to each element in the vector using it's name with a single set, or double set of brackets to get the value, or the value and name, respectively.

```
bb['hello']
```

```
## hello
##     6
```

```
bb[['hello']]
```

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

Fun.

lists

Indexing on lists is similar to vectors. A huge difference though is that lists can be nested. So there could be infinite things within each slot of a list.

For example, let's say we have the nested list from above `mylist`

```
mylist <- list(foo=1, bar=list("a","b","c"))
```

We can index to the first item in the list, including it's name, by

```
mylist[1]
```

```
## $foo
## [1] 1
```

Or equivalently

```
mylist['foo']
```

```
## $foo
## [1] 1
```

And get just the value by using two `[`

```
mylist[[1]]
```

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

Or equivalently

```
mylist[['foo']]
```

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

And get the second item in `mylist` by

```
mylist[2] # or mylist['bar']
```

```
## $bar
## $bar[[1]]
## [1] "a"
##
## $bar[[2]]
## [1] "b"
##
## $bar[[3]]
## [1] "c"
```

```
mylist[[2]] # or mylist[['bar']]
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] "c"
```

And get to the individual elements within `bar` by

```
mylist[[2]][1]
```

```
## [[1]]
## [1] "a"
```

And so on to get to get to what you need.

There are a number of convenience functions to make working with lists easier, but you can learn about those later.

data.frame and matrix

Indexing on a `data.frame` and `matrix` is similar. Both have two things to index on: rows and columns. Within `[,]`, the part before the comma is for rows, and the part after the comma for columns. So if you have a data.frame `iris` in R,

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

you can index to the third row and second column by doing

```
iris[3,2]
```

```
## [1] 3.2
```

You can also use names to index if you have named rows or columns. For example,

```
iris[2,"Species"]
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

You can also use the $ symbol to index to a column, like

```
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

Functions

Cats are the type of feline to love functions. Functions make your life easier by allowing you to generalize many lines of code, and avoiding repeating yourself. Functions make your work tidier - just like cats like it.

Functions are written like this

```
foo <- function(){
  writeLines("I hate dogs")
}
```

After defining this function we can then call it later like this

```
foo()
```

```
## I hate dogs
```

Yay! Dumb dogs.

The `foo` function was pretty simple. We can also pass in parameters to the function.

```
foo <- function(mess){
  writeLines(mess)
}

foo("I hate dogs")
```

```
## I hate dogs
```

And set parameters to default values.

```
foo <- function(mess = "I hate dogs"){
  writeLines(mess)
}

foo()
```

```
## I hate dogs
```

```
foo("Well, I hate most dogs, but I like the one in my house")
```

```
## Well, I hate most dogs, but I like the one in my house
```

Generally, if you are writing more than 3 lines of code to do any particular task you may as well write a function to do that task, making it reusable and (hopefully) more general.

Using packages

Since you're a cat, you can think of packages like boxes that you put a bunch of code in. Since you are putting code in this box you probably don't want to sit in it :). These boxes generally hold a similar set of functions (see functions above). A package allows you and others to

- Easily install and load the code
- Incorporate documentation
- Lessen conflicts with functions in other packages

Most people that make R packages share them on site on the interwebs called CRAN (don't worry about what it stands for) here CRAN.

The humans behind CRAN have done a good job making sure that in most cases packages you install from CRAN will work on your computer.

Installation is super easy. Do `install.packages("package_name")`, where `package_name` is the name of the package you want to install. Remember that the package name is case sensitive! Or if you're using RStudio you can go to the *Packages* pane.

Once the package is installed you have to load the package in to your R session. That's easy too! Do `library('package_name')`, or if you're in RStudio go to the *Packages* pane.

Note: Package creation is bit out of scope for this site, but Hadley has made it much easier with devtools.

No no's for cats using R

There are a few R gotchas to avoid cat friends.

- `attach()`: Don't use it. Just, don't.
- When doing `library()` or `require()` use the package name in quotes as package name without quotes is sugar, but can cause problems.

Do do's for cats using R

There are a few things I recommend when using R.

- Do combine code and text with `Markdown` or `LaTeX` to have reproducible documents, using `knitr`.
- Do share your code.
- Do ask lots of questions on StackOverflow (use the `[r]` tag), Twitter (does this need saying), etc.

Data from the web

Install `cowsay`

```
install.packages("devtools")
library("devtools")
install_github("sckott/cowsay")
```

Now let's get a cat fact!

```
library("cowsay")
say("catfact", "cat")
```

```
##
##
##  -----
##  Cats respond better to women than to men, probably due to the fact that women's voices have a hi
##  ------
##     \
##      \
##                 \‘*-.
##                  )  _‘-.
##                 .  : ‘. .
##                 : _   ’
##                 ; *‘ _.   ‘*-._
##                 ‘-.-’          ‘-.
##                   ;       ‘       ‘.
##                   :.       .       \
```

```
##                    .\  .   :   .-'   .
##                    '  `+.;  ;  '      :
##                    :  '  |    ;       ;-.
##                    ; '   : :`-:     _.`* ;
##                 .*' /  .*' ; .*`- +'  `*'
##                 `*-*   `*-*  `*-*'
##
```

A little explanation is in order me thinks. There are a few things going on in the last thing we just did. The `say` function looks like sorta like this:

```
say <- function(what, by, type){
  <== some ascii art ==>
  url <- "http://catfacts-api.appspot.com/api/facts?number=1"
  what <- fromJSON(url)$facts
  message(sprintf(by, what))
}
```

The first line is a bunch of ascii characters to make a cat. The second line defines the url for the cat facts API. The third line retrieves one cat fact from the cat facts API. And the fourth line prints the messages with a cat.

> But what is an API? I'm a cat, I only drink water or milk (preferably milk) - but at least I've heard of an IPA. What the rat's ass (yum) is an API.

Okay, here goes. An API stands for Application Programming Interface. It's just a set of instructions for two computers to talk to each other. It's sorta like if you run into another cat and if you both knew beforehand a lot about each other, you would have a sense for how to behave - if you don't know each other, then best of luck to you Mr. Tickles.

Reading

After this basic intro you'll want to head over to:

- XXXXX

And for even more advanced R:

- Advanced R, by Hadley Wickham

Cat's love R

License