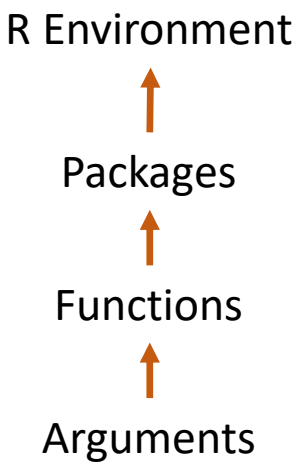
# R: Some Helpful Vocabulary

We often think about a data analysis program as a single program that is developed by one company and is self-contained. You install the program once and it has everything you need to run it. The various components of the program are well-integrated and the conventions are consistent across the different parts of the program.

R was built as an open-source platform that allows people to easily collaborate. Anyone can rapidly develop complex analysis using a very flexible programming framework. This analysis is shared with other users either through sharing code (scripts), or through packaging that code and sharing them as programs (packages).

As a result, it is more useful to think about R as an operating system rather than a program. Just like Windows makes it possible for your computer to run a bunch of programs that were not written by Microsoft, R makes it possible to run a bunch of programs that were developed by members of the community. In this way R is as much a community as it is a program. The “R Environment” is a set of protocols that people use to share their work.  Here is some vocabulary that will help you make sense of the R paradigm:




## The R Environment

In software terms, an environment is the term used for the things that are needed to run an application. On a laptop, the environment for a specific desktop application might include the operating system, a database, and a compiler. R is called an environment because it has all of the elements necessary to run programs written for R.

## Package

R programs are called “packages” and they are loaded by the “library” command. The term comes from the fact that each package (program) is comprised of a bunch of functions. They are organized into a library, and this library is shared by packaging it all together into a single entity.

## Function

A function is a small program that does one specific task or calculation. For example, it could be a program that calculates the average of a bunch of numbers. It could be a program that changes the temperature in Celsius to the temperature in Fahrenheit. It could also be a program that creates a graphic. Any task that you do in R is accomplished by a function, and your analysis will consist of a bunch of functions used to manipulate your data.

## Argument

Each function is like a recipe that needs ingredients in order to work (see the example below). The ingredients are either data that you give to the function, or specific parameters it needs in order to run. For example, if you want to calculate a mortgage payment you need to know the interest rate (a parameter), otherwise the calculation will not be possible.

## Script

A script is a short program that accomplishes some analysis using several functions.

## Integrated Development Environment (IDE)

A software application that provides a comprehensive set of facilities needed to effectively develop and test code. R Studio is one of many IDE's used by R programmers. It is unique in that it was developed specifically for R.

## CRAN

Comprehensive R Archive Network, or CRAN, is the online repository that houses all packages written for R. There are currently over 7000 packages available for download. You can browse all packages here: http://cran.r-project.org/web/packages/

## Mirror

A local server used to download R packages.

## Path

The address of a file in the directory structure.

## Object-Oriented Programming

A programming paradigm that breaks chunks of code into "objects" to make them re-usable and robust.

## Class

The type of object in the R environment.

## Mode

The underlying data structure of a class.

# R Style Guide

## Naming Conventions

### File Names

The file name should end in .R or .r

Make them meaningful

```
# GOOD

Mackey Clustering Algorithm.R


# BAD

homework.r

ty2.r
```

If the file is a function that will be sourced, name the file the same as the function

```
# Function

plotInColor <- function( x, y )
{
    plot( x, y, col=factor(x) )

    return( NULL )
}


# File name

plotInColor.r
```

## Variable Names

There is agreement on some naming conventions, such as use nouns to name variable and datasets; use verbs to name functions.

There is disagreement, though, over the specific syntax style for functions in R. You can see this across conventions used in each package. So these rules are far from universal, but it is useful to get in the habit of distinguishing between data names and function names in your own code. My suggestion is to use all lowercase letters with words separated by periods for names of variables and datasets:

```
# GOOD
my.data.frame <- matrix( rnorm(1000), nrow=100 )
dat.atl <- dat[ dat$FIPS %in% atl.fips , ]
lm.01 <- lm( y ~ x )

# BAD
MyDataFrame  # use camel caps for functions
my_data_frame  # separate by periods
My.data    # don't use upper case
01.lm <- lm( y ~ x )  # R won't allow names to start with num's
```

## Function Names

Use verbs that describe what the functions do. Use camelCaps to differentiate functions from datasets in your code.

```
# GOOD
makePretty <- function( ) { … }
subsetMyData <- function( ) { … }

# BAD
make.pretty <- function( ) { … }    # use camelCaps
prettyGraph <- function( ) { … }    # should be a verb
```

# Spacing

Place spaces around all operators: = , <- , + , - , / , > etc.

Place spaces after commas, but not before

Places spaces after the parenthesis in a function and after brackets [ ]

```
# GOOD

x <- intersect( z1, z2 )

table.01 <- tapply( y, x1, mean, na.rm=T )

# BAD

x<-intersect(z1,z2)

table.01 <- tapply(y,x1,mean,na.rm=T)
```

It doesn't hurt to add an extra space before a comma in the middle of a data frame or matrix bracket set to make it clear that there are two indices.

```
# GOOD

dat[ 1:10 , 2:5 ]

sub.dat[ dat$EIN == i , ]


# BAD

dat[ 1:10,2:5 ]

sub.dat[dat$EIN==i,]
```

# Indentation

## Loops and Functions

Use a three-space indentation to delineate a loop or the body of a function. The rule is recursive. Put curly brackets on their own line.

```
for( i in 1:10 )
{

   print( i + 1 )

   for( j in 1:5 )
   {
      print( i*j )

      count <- count + 1

   } # end of j loop #

} # end of i loop #
```

## Lists of Arguments

Also use indentations to separate a long list of arguments in a function:

```
plot( x, y,
      main  = ”This is my graph title”,
      xlab  = ”The X Label”,
      ylab  = “The Y Axis”,
      color = col.vector
    )
```

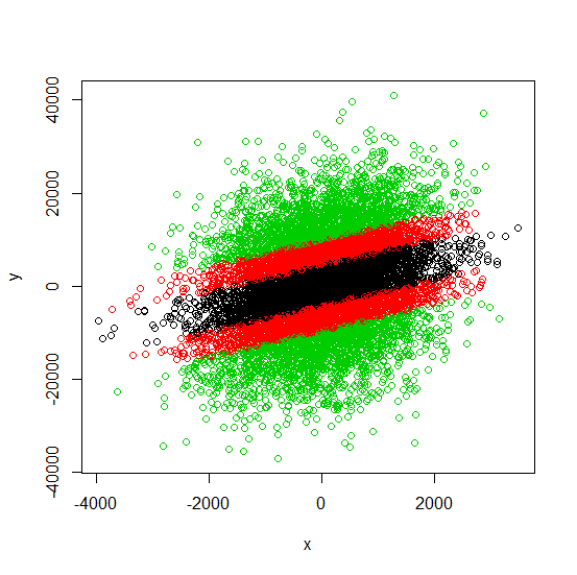
# Documenting Functions

A function is an input, output device. Directly above the function describe the input – what arguments the function accepts (including the data type of each), what the function does, and the output – what the function returns.

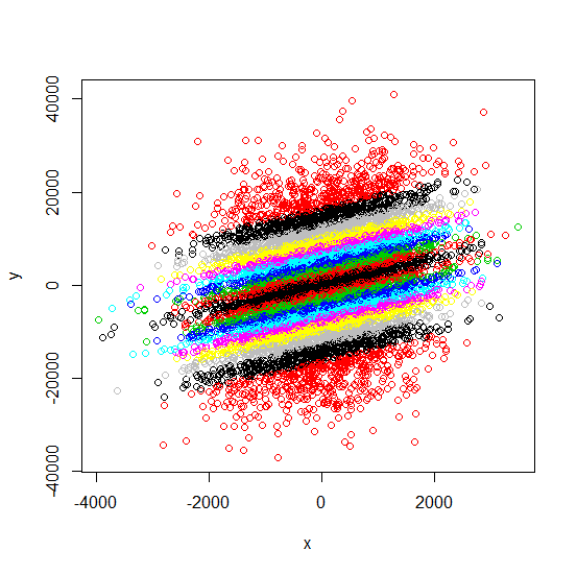
It is good practice to always include a return call in a function, even if it is just NULL. If the end of a function is reached without a return call R the value of the last evaluated expression is returned. So if you want the function to return nothing you need a NULL return.

```
# ====================================================
#
#   The 'plot.residual.colors' function.
#
#   Plots an x and y variable and adds color to the
#   data points to indicate distance from the
#   regression line.
#
#   Arguments:
#     x       = a vector of numbers
#     y       = a vector of numbers
#     res.col = number of color bands on the plot
#
#   Returns:  NULL
#
# ====================================================


plot.residual.colors <- function( x, y, num.cols=3 )
{
   m01 <- lm( y ~ x )
   cats <- cut( rank( abs( m01$residuals ) ), num.cols )

   plot( x, y, col=cats )

   return( NULL )

} # end of function #
```

```
plot.residual.colors( x, y )
```




```
plot.residual.colors( x, y, 10 )
```



# Writing Scripts

A script is a short program for data analysis. It is important to organize your scripts in a consistent manner. There are several things that should be included at the beginning of each script:

(1) Documentation which can include the purpose, author, copyright info, and version of the script

(2) Load all packages needed for the program

(3) Source any custom functions needed for the analysis

(4) Declare any universal variables (constants)

(5) Set any session options

```
# ======================================================
#
#   Step 1 of analysis of tree frog growth rates.
#   By: Keyser Söze
#   Last updated:  May 1, 2013
#
#   Merges data on tree frogs with ecological data
#   from the observation sites.
#
# ======================================================

library(foreign)
library(ggplot2)

source("./Functions/estimateByGrid.R")

options( digits=2 )


# Start your script here
```

See below for instructions in how to organize your scripts into a directory structure that will streamline your work-flow.

# Miscellaneous Operators

## Assignment

R uses a specific throw-and-catch convention in order to make it easier to interact with data without writing a lot of print functions. If you type the name of an object or evaluate an expression, the default behavior is to print the object or result. If you want to save the changes, you need to include a 'catch' statement, i.e. assign the results to a new variable. Assignment in R is done through the **<-** operator. Technically the **=** operator works, but it is discouraged for assignment. Instead, it should be used for arguments inside of functions.

```
# GOOD

x <- 10

10 -> x  # this is not as common, but allowed

plot( x=edu, y=income, main="Education and Income" )

      # use the = operator for arguments in a function


# BAD

x = 10  # this works but is discouraged

x < - 10  # be careful! this reads, x is less than -10
```

# Quote Marks

In R you reference objects (datasets or functions that are loaded in your environment) by name directly, and you reference arguments and strings using quotation marks. R allows you to use single or double quotation marks.

Double quotes are encouraged in order to avoid a subtle bug that can creep into your code when working across platforms. The single quote ' is not the same as the prime ` even though they look similar. Some text processors will replace quotes with primes for style purposes. R can interpret the quotes, not the prime symbol, though.

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

> x           # prints the object as expected
[1] 1 2 3

> "x"         # prints the character "x" as expected
[1] "x"

> "x"         # pretty quotes not recognized

Error: unexpected input in """

> 'x'         # single quotes are interpreted same as double
[1] "x"

> `x`         # unexpected behavior - ignores primes
[1] 1 2 3
```

~~~

## Examples of Other Style Guides

http://csgillespie.wordpress.com/2010/11/23/r-style-guide/

http://stat405.had.co.nz/r-style.html

http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

## Resources

News and tutorials from the R community:

http://www.r-bloggers.com/

NY Times blog on producing graphics with R:

http://chartsnthings.tumblr.com/

Quick-R, a great guide to basic analysis with examples:

http://www.statmethods.net/

A reference card for common R functions:

http://cran.r-project.org/doc/contrib/Short-refcard.pdf

Stack Overflow thread on R:

http://stackoverflow.com/questions/tagged/r

R packages sorted by topic:

http://cran.r-project.org/web/views/

Color guide:

http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

Great Graphics Blog

www.flowingdata.com
~~~

## Example of Functions and Arguments

```
### A simple mortgage calculator

   calcMortgage <- function( years, APR, principal )
   {

      months <- years * 12
      int.rate <- APR / 12
      monthly.payment <- ( principal * int.rate ) /
                         (1 - (1 + int.rate)^(-months) )
      return( monthly.payment )

   }




#  Calculate monthly payments for a 30-year mortgage   #
#  on a $100,000 loan at a 5% APR interest rate:       #

calcMortgage(
                 years=30,
                 APR=0.05,
                 principal=100000
             )

[1] 536.8216  # output from the console



### Does not run:

calcMortgage( principal=250000 )

# here the function does not know the term of the loan or rate



### 'throw' plus 'catch' in order to save the value

my.payment <- calcMortgage( years=30, APR=0.05, principal=100000 )

>my.payment
[1] 536.8216
```

```
###  Example of a function with some default values provided


   calcMortgage <- function( years=30, APR=0.05, principal )
   {

      months <- years * 12
      int.rate <- APR / 12
      monthly.payment <- ( principal * int.rate ) /
                         (1 - (1 + int.rate)^(-months) )
      return( monthly.payment )

   }



# the function now runs, because if you don't provide it with
# an argument it assumes you want to use the default values.

calcMortgage( principal=250000 )
[1] 1342.054



# you can over-write default values by supplying an argument

calcMortgage( years=15, principal=250000 )
[1] 1976.984
```