CSHS-hydRology: Intro to R Webinar

R. Chlumsky

May 15, 2019

Intro to R Tutorial

This beginner R tutorial covers the following topics:

- Navigating RStudio
- Basic R Syntax
- Data types in R
- Help in R
- Installing and Exploring Packages
- Data checking and sample data sets
- Creating plots
- Basic programming structures
- Building functions

Please work your way through the tutorial and ask any questions you may have!

How this document works

This document was generated using R Markdown. This is a great tool for code transparancy and data analysis, because the code blocks, code outputs, and your comments are "knitted" into a single document!

This document will have a number of headings, text blocks (like this one), and code blocks that are run as the document is compiled. This allows you to see the script and output in one document (plus the code gets run and checked for errors as the document is made, which helps to reduce typos in the document code sections).

```
# this is a comment in R. It is preceeded by an octothorp (or pound symbol,
# or hashtag, depending on your generation).
# A comment shows up in this dark orange colour

print("This is a line of code with text in it, preceeded by ## in the output printout.")
```

```
## [1] "This is a line of code with text in it, preceeded by ## in the output printout."
```

With this document, you can see the code block and the output in one document. Neat!

Basic R Syntax

Basic Arithmetic

Let's try some basic math, that shows R can be an overly fancy calculator. We see common signs like addition, multiplication, and division. In base-R or a script in RStudio, you can place your cursor in one line and press CNTL+ENTER to run that one line (CMD+ENTER for Mac). Additionally, you can highlight multiple lines, or all with CNTL+A and then CNTL+ENTER. The Run button in the upper right of your interface will run either the selected or highlighted lines as well.

Note: when you see the pount or "hashtag" next to a line of code, what follows to the right is a text comment that does not affect the code. Commenting is an important part of remembering what complex lines of code are meant to do!

```
1+1
```

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

```
2*2
```

```
## [1] 4
```

```
4/3
```

```
## [1] 1.333333
```

If you're completely new to coding or "scripting", you may wonder what the main gist of R is. R, like other languages including C++, Java, Python, and Matlab, is an *Object-Oriented Language*. For our purposes, that means we will be storing data within objects of various shapes, sizes, and properties, and using **functions** to operate on those objects to perform analyses.

Let's explore R's simplest object type, an **atom**. R is not strict in its definition of data types, i.e. you can declare and re-assign the data types to variables at your whim.

One final note. The sign you see below (<-) *assigns* the value from one end of the arrow to the object at the point of the arrow. This method is directional, and comes in handy, but often you will see the equal sign (=) as a left-only equivalent.

```
a <- 1
a
```

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

```
a = 1
a # see that <- and = gave the same result here
```

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

```
5 -> a
a
```

```
## [1] 5
```

```
b <- a+1 # This is our first example of
b
```

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

Intro to Functions

You've seen data stored in objects. So how can we do more than just perform arithmetic on these objects? R operates by loading and creating objects and modifying them or parts of them with functions. Functions have the following format: **function(x)** Where x is an object, and "running" this function will produce some output.

To understand how functions work, let's first create our own. Creating your own custom functions is a way to streamline your research and perform ever-faster analyses. We'll work with R's built-in functions in a moment.

Here is a function that we will name "squared". The input is some value, x, and the output is x squared (x^2).

```
squared=function(x){x^2}
squared(3)
```

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

```
squared(a)
```

```
## [1] 25
```

Note: The squiggly brackets "{ }" Always surround the code being performed by a function. They **cannot** be interchanged with the parentheses "( )". Parentheses are used to define the input to a function, but they can also be used to specify order-of-operations in arithemtic or complex calculations.

Recall, a had a value of 5, so the function returns a value of 25. Like we did with arithmetic functions, we can assign the output of a function to a new object.

```
c=squared(b)
c
```

```
## [1] 36
```

```
# optional parameter
exponentit <- function(a,b=2) {
  a^b
}
exponentit(3) # no b-value supplied, taken as the default 2
```

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

```
exponentit(3,3) # b-value supplied as 3, 3 overrides the default
```

```
## [1] 27
```

Data Exploration and Data Types

Here are some commands to check what type of data you are dealing with, and help you convert between different types.

```
# what class of variable is it?
class(b)
```

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

```
# what 'typeof' value is b?
typeof(b) # 'double' is a type of number (numeric value)
```

```
## [1] "double"
```

```
b <- "hello, world"
b
```

```
## [1] "hello, world"
```

```
# this has now changed from numeric/double
typeof(b)
```

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

```
# convert between data types
as.numeric("501") +1
```

```
## [1] 502
```

```
round(5.6) # round to integer
```

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

```
floor(5.6) # round down to integer
```

```
## [1] 5
```

```
# as.numeric(b) # will produce an error
# as.character(a) + "_" # will produce an error
# as.character(a) && "_" # will produce an error
paste0(as.character(a),"_") # one proper way to combine strings
```

```
## [1] "5_"
```

Now that we've started re-assigning values to variables, it's useful to recall their new values. On the upper-right portion of your screen in RStudio, the **Environment** tab shows the values of all user-defined variables, and it even reminds us of our user-defined function.

Working with Text

R has a large capability to read, write, and modify strings of text. Here are just two examples. We will come back to working with text once we start generating labels for plots.

```
# write a statement with any format - substitute values from any type
paste0(as.character(a)," coconuts in my pocket")
```

```
## [1] "5 coconuts in my pocket"
```

```
sprintf("I have $%.2f in my pocket, now I am %s!",a,"rich")
```

```
## [1] "I have $5.00 in my pocket, now I am rich!"
```

Vectors and Matrices

Recall, a single valued varible, such as the ones above, is called an *atom*. Multiple atoms come together to form a row of values, called a **vector**. Multiple vectors form a **matrix** or **data frame**.

c() is the concatenate function, taking multiple objects separated by commas and creating a new vector. We saw above how we can use addition or multiplication on an atom. We can also perfom a single operation on all values in a vector. Let's perform apply arithmetic *and* our custom "squared" function on the new vector **a**.

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

```
## [1] 1 2 3 4 5
```

```
b <- a/2
b
```

```
## [1] 0.5 1.0 1.5 2.0 2.5
```

```
b=squared(a)
b
```

```
## [1]  1  4  9 16 25
```

Getting Help on Commands

You'll see below several functions that are new to you. A useful tool is the built-in **Help** database built into R and RStudio. For the function *matrix*, look up its documentation. The help() function can show you the Description and Usage of a function. Putting "?" before a function does the same. If you don't quite know the name of a function, two "?" question marks will search for all matrix-related topics. Finally, if you place your cursor within the function name and press **F1** on your keyboard, the documentation will again come up.

```
rep(1,5)
```

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

```
seq(1,4)
```

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

```
# get help on command, like the variance function 'rep()'
help(rep)
```

```
## starting httpd help server ... done
```

```
?rep
?seq

# search for functions by name or keyword
??variance

# see R code directly
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x0000000017c6d7e8>
## <environment: namespace:stats>
```

```
sqrt
```

```
## function (x)  .Primitive("sqrt")
```

For more resources, check out the CRAN website (https://cran.r-project.org/) for information, documentation, etc. StackOverflow and similar forums are also extremely helpful for R!

Use help to figure out the arguments to the functions “rep” and “seq”.

```
c(a,rep(1,5))
```

```
##  [1] 1 2 3 4 5 1 1 1 1 1
```

```
seq(1,10,by=2)
```

```
## [1] 1 3 5 7 9
```

Subsetting Vectors

Now that we've started working with objects of increasing size, it's important to know how to extract specific values. To do this, we use the square brackets "[ ]". They are not to be confused with () or {}, which each have their own special uses, as discussed above. A useful tool when examining large objects is the **length()** function.

```
# sequence from 1 to 30, jumping by a value of 2 (instead of the default 1)
a <- seq(1,30,by=2)
a
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29
```

```
length(a)
```

```
## [1] 15
```

```
# subset by location
a[1] # first value in the vector
```

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

```
a[c(1,2)] # first and second value in the vector
```

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

```
a[c(1,2,3)] # note the use of c() to specify multiple values
```

```
## [1] 1 3 5
```

```
1:3 # the colon ":" is a useful operator that operates similar to seq(by=1)
```

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

```
a[1:3]
```

```
## [1] 1 3 5
```

```
a[-1]  # all values without the first one
```

```
##  [1]  3  5  7  9 11 13 15 17 19 21 23 25 27 29
```

Using Booleans

Boolean operators refer to greater than (>), less than (<), or equal to (==). Other operators or greater than *or* equal to (>=), and not equal to (!=)

```
1>3
```

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

```
1<3
```

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

```
1<=3 # less than or equal to
```

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

```
1!=3
```

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

```
1==1
```

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

Note: <= looks very similar to <- but they are nothing alike.

We can use Booleans to perform subsetting operations.

```
a>5 # which values are greater than 5?
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE
```

```
a[a>5] # same as above, but returns the values of a that meet the critera
```

```
##  [1]  7  9 11 13 15 17 19 21 23 25 27 29
```

```
a[(a >= 3) & (a < 20)] # multiple conditions for subsetting
```

```
## [1]  3  5  7  9 11 13 15 17 19
```

```
subset(a, a>=3 & a<20) # same operation with the subset function
```

```
## [1]  3  5  7  9 11 13 15 17 19
```

```
a[which(a>=3 & a<20)] # same operation as above with the which function
```

```
## [1]  3  5  7  9 11 13 15 17 19
```

```
which(a>=3 & a<20) # returns the index values *which* meet the criteria specified
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

2D Data: Matrices and Data Frames

Now that we've worked with 1-Dimensional vectors, let's build up to two dimensions.

Notice the new function "dim()", which gives us the dimensions of the object, like "length()" did for vectors.

```
a=1:3
b=squared(a)
```

As an aside, notice that our **squared** function works on a vector as well as single values. When the **squared** function is called for a vector, it applies the function across all the values by default. Here lies the power of R as well as the source of a lot of frustration for new useRs!

Returning to data frame and matrix formation...

```
c=data.frame(a,b)
c
```

```
##   a b
## 1 1 1
## 2 2 4
## 3 3 9
```

```
typeof(c)
```

```
## [1] "list"
```

```
dim(c) # returns rows, then columns
```

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

```
# b <- matrix(base value for matrix, number of rows = 4, number of columns = 4)
d <- matrix(NA,nrow=4,ncol=4)
d
```

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

```
typeof(d)
```

```
## [1] "logical"
```

```
dim(d)
```

```
## [1] 4 4
```

Here we've built a dataframe, "c", with columns "a" and "b". "d" is an emptry matrix, with 4 rows and 4 columns. While both data types are two dimensional, matrices are easier to operate across both dimensions. Data frames are mostly used when you are operating across columns.

Subsetting both of these data types works the same. Like with vectors, we use the square brackets "[ ]", but instead of one value, we list the desired row and column, separed by a comma.

```
c[2,2]
```

```
## [1] 4
```

```
d[2,2]
```

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

```
c[2,] # leaving an entry black will return the entire row or column specified
```

```
##   a b
## 2 2 4
```

```
c[,2]
```

```
## [1] 1 4 9
```

```
d[2,]
```

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

```
d[,2]
```

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

An important distinction: while "calling" a row of a dataframe will return a mini dataframe with just the one row, a row of a matrix will return a vector. This shows the flexibility of matricies.

```
d[,2] <- rep(2,4)
d[3,] <- seq(1,4)
d
```

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

```
d[3:4,1:2] # subset across multiple rows, columns
```

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

```
## [2,]   NA    2
```

```
# sequence built with the seq() function
seq1 <- seq(1:6)
a <- matrix(seq1, 2)
a
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Data frames have an additional method of subsetting, by using the names of the columns within them. The "$" sign can be used to directly call the named column.

```
names(c)
```

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

```
c$a
```

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

```
c[,"a"]
```

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

```
c["b"]
```

```
## [1] 1 4 9
```

You can also direclty

Running Commands

In R, commands are run with round brackets. In this example, the rnorm function is called.

```
a <- rnorm(n = 100,mean=0,sd=2) # generate a normal distribution
mean(a)
```

```
## [1] 0.04674178
```

```
sd(a)
```

```
## [1] 2.075187
```

Explore More Packages!

R has **thousands** of packages to offer, which is one of the main advantages of getting to know R. This section will show you how to easily install and get to know new packages.

Installing and exploring a new package

```
# install ggplot2 with dependencies from CRAN (internet connection required)
install.packages("ggplot2")

# load the library into your R session
library(ggplot2)

# explore the package contents
head(ls("package:ggplot2"),40)
```

Some Data on Handling Data

Here we will work with the ggplot2 library, which has a sample dataset directly included, called **diamonds**. Let's explore this diamonds dataset with some commands and see what we are dealing with.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.4.4
```

```
data(diamonds)  # load data frame from ggplot2 package
head(diamonds)  # preview data
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

```
class(diamonds)   # class of an object
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
typeof(diamonds)  # type of object
```

```
## [1] "list"
```

```
names(diamonds)   # names inside the object
```

```
##  [1] "carat"   "cut"     "color"   "clarity" "depth"   "table"   "price"
##  [8] "x"       "y"       "z"
```

```
str(diamonds)   # view structure of diamonds
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

```
summary(diamonds) # summarize dataset
```

```
##      carat               cut        color        clarity
##  Min.   :0.2000   Fair      : 1610   D: 6775   SI1    :13065
##  1st Qu.:0.4000   Good      : 4906   E: 9797   VS2    :12258
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194
##  Mean   :0.7979   Premium   :13791   G:11292   VS1    : 8171
##  3rd Qu.:1.0400   Ideal     :21551   H: 8304   VVS2   : 5066
##  Max.   :5.0100                     I: 5422   VVS1   : 3655
##                                     J: 2808   (Other): 2531
```

```
##      depth           table           price             x
##  Min.   :43.00   Min.   :43.00   Min.   :  326   Min.   : 0.000
##  1st Qu.:61.00   1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710
##  Median :61.80   Median :57.00   Median : 2401   Median : 5.700
##  Mean   :61.75   Mean   :57.46   Mean   : 3933   Mean   : 5.731
##  3rd Qu.:62.50   3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540
##  Max.   :79.00   Max.   :95.00   Max.   :18823   Max.   :10.740
##
##        y               z
##  Min.   : 0.000   Min.   : 0.000
##  1st Qu.: 4.720   1st Qu.: 2.910
##  Median : 5.710   Median : 3.530
##  Mean   : 5.735   Mean   : 3.539
##  3rd Qu.: 6.540   3rd Qu.: 4.040
##  Max.   :58.900   Max.   :31.800
##
```

```
nrow(diamonds)  # number of rows (lots!)
```

```
## [1] 53940
```

```
ncol(diamonds)  # number of columns
```

```
## [1] 10
```

```
# subset particular data from data frame
head(diamonds$carat)
```

```
## [1] 0.23 0.21 0.23 0.29 0.31 0.24
```

```
mean(diamonds$price)
```

```
## [1] 3932.8
```

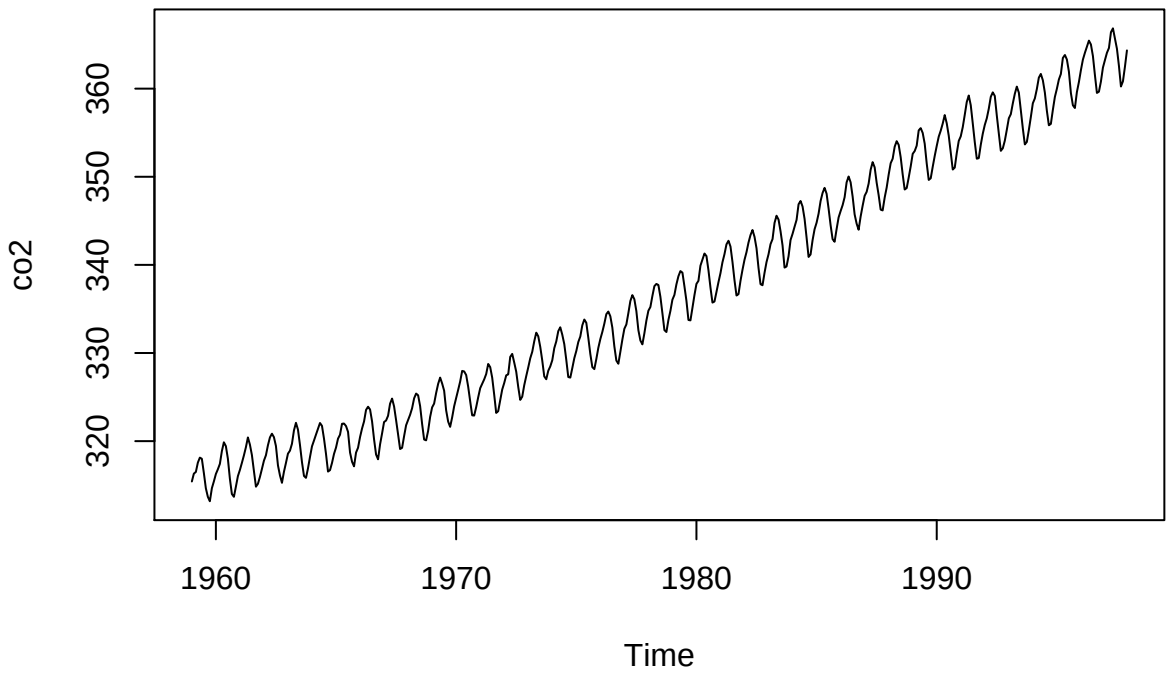
```
mydb <- diamonds[diamonds$price > 3000,]
```

R also has a large number of built in sample datasets, which can be viewed with the **data()** command. These are useful for testing functions and making reproducible examples for other users. Try taking a look at the LakeHuron and co2 datasets.

```
data() # view the full set of sample data sets

# examine and make a quick plot with the CO2 sample observation data set
head(co2)
```

```
## [1] 315.42 316.31 316.50 317.56 318.13 318.00
```

```
plot(co2) # more on plotting later!
```




```
# summarize the Lake Huron sample data set
summary(LakeHuron)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   576.0   578.1   579.1   579.0   579.9   581.9
```

Explore the CSHS-hydRology package

The CSHS-hydRology package is created by and for Canadian hydrologists to address some of the challenges unique to Canadian landscapes and Canadian hydrology. The package is open source and available for use from Github. To learn more about the package, let's use R to find the commentary online (this will open a web broswer with your computer's default broswer).

```
browseURL("https://www.usask.ca/hydrology/papers/Anderson_et_al_2018.pdf")
```

The hydRology package can be installed from Github directly with the **devtools** library. Note that the devtools package may take a few minutes to install.

```
install.packages("devtools")
library(devtools)
install_github("CSHS-hydRology/CSHS-hydRology")
library(CSHShydRology)
```

If you want to find out more about the package itself directly within R, try using the help functions to find more information (this information is embedded directly in the package, as with all R packages). This will open a new page in the **Help** tab.

```
?`CSHShydRology-package`
```

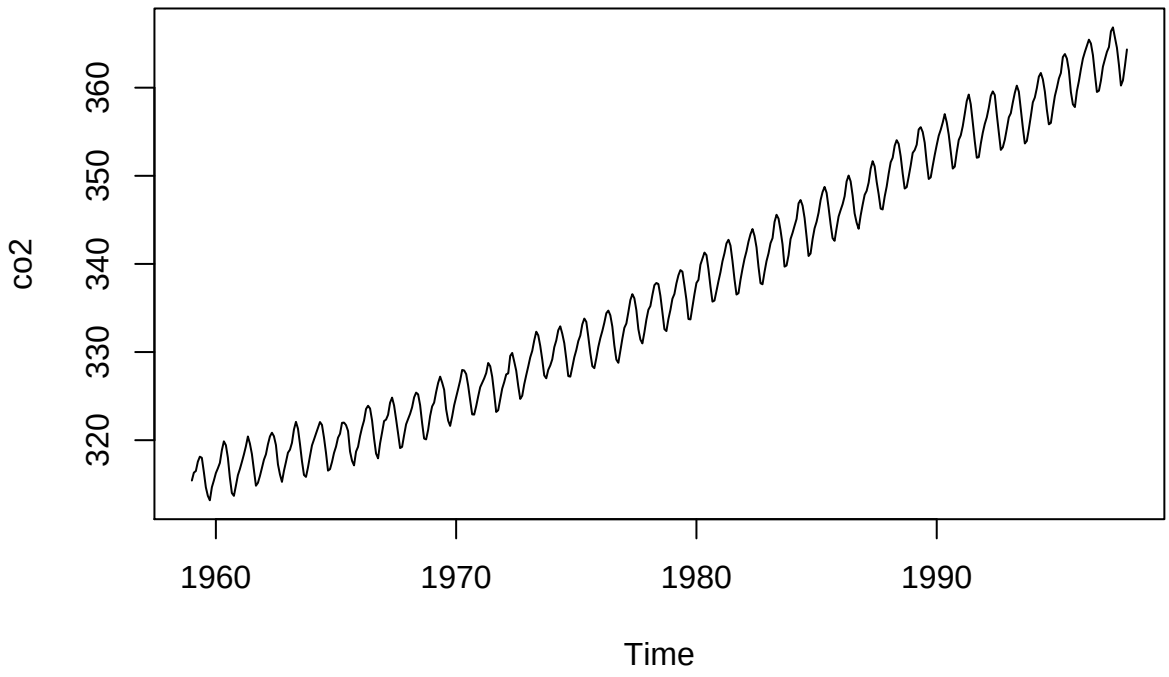
If we want to check the list of functions in the CSHShydRology package, we can use the following command.

```
ls("package:CSHShydRology")
```

Explore the plot function

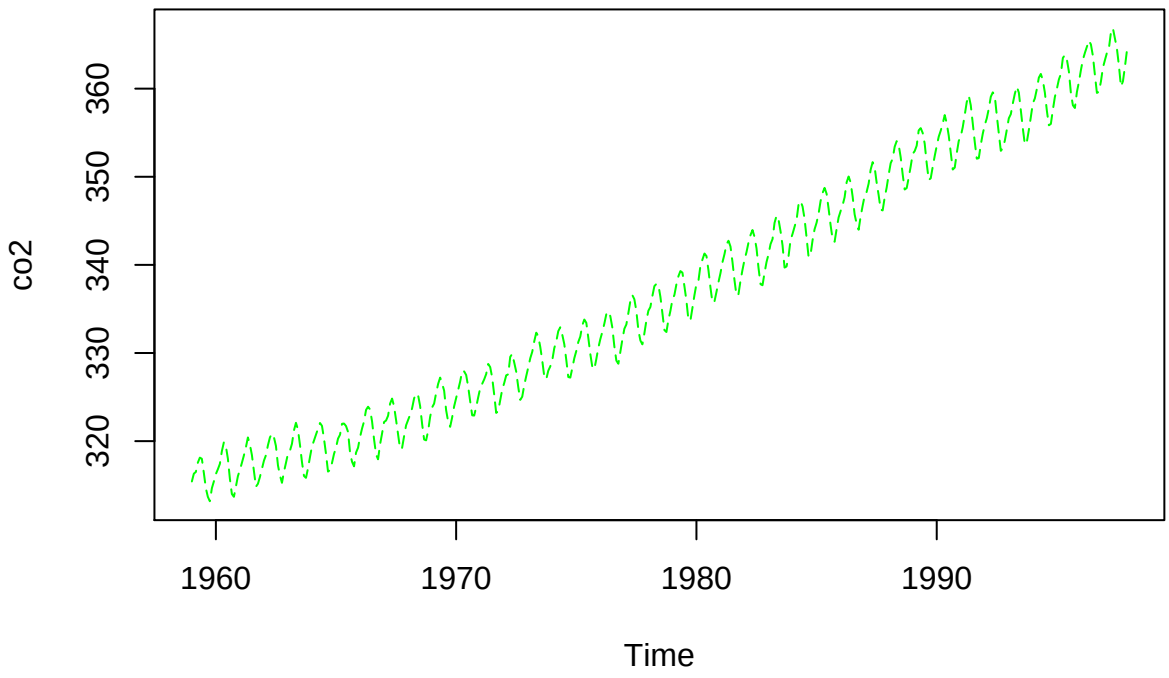
What fun is data if you can't visualize it? Let's revisit the CO2 dataset and recreate our plot.

```
plot(co2)
```

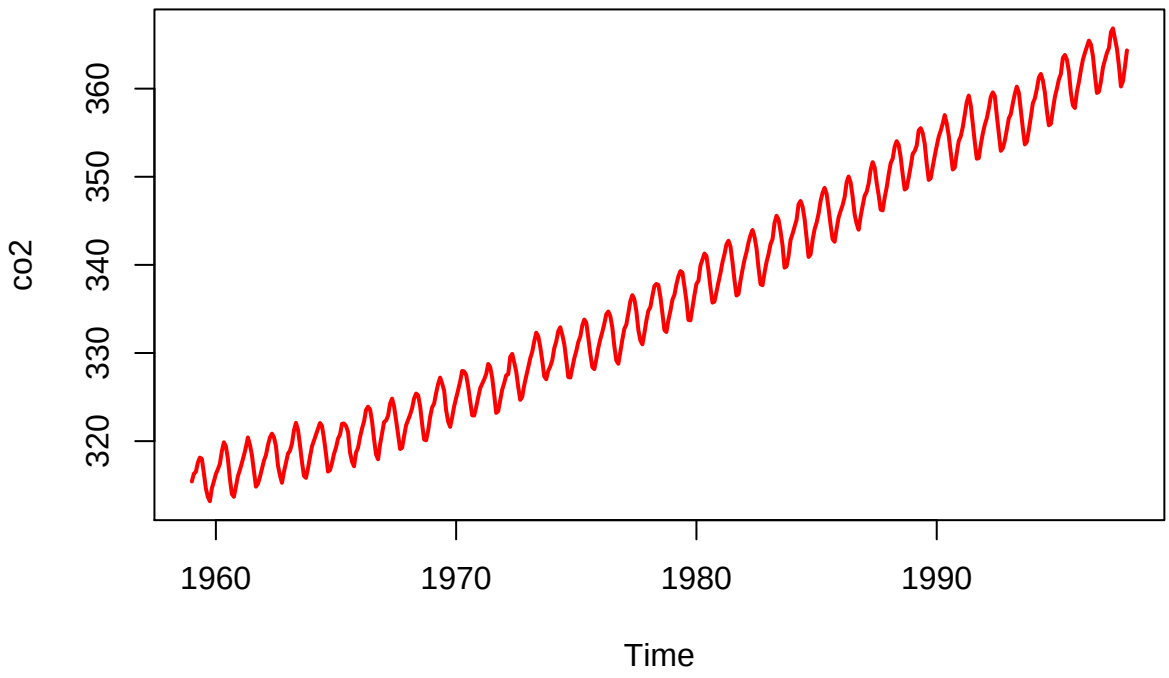


This plot is ok, but we can do better. Let's try to adjust the colour and style of the plot.

```
plot(co2, col='green',lty=5)
```

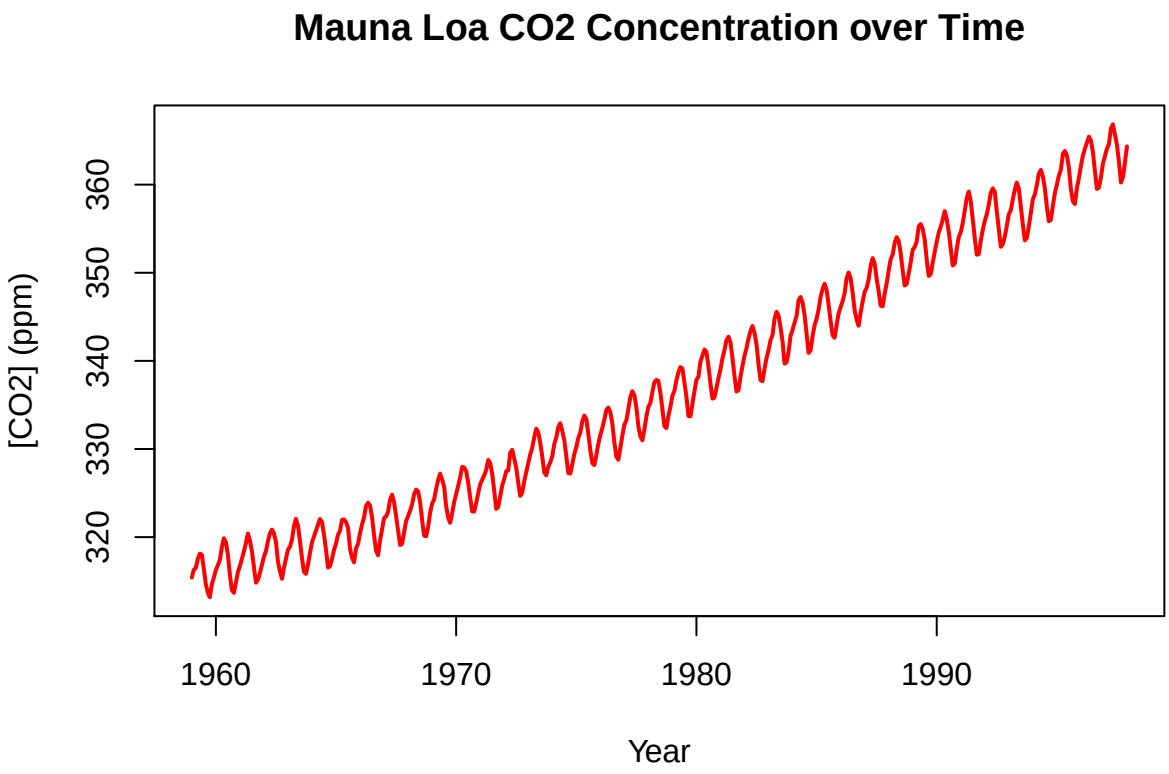



```
plot(co2,col='red',lwd=2)
```



What about the axes labels? We can override the defaults and add more prescriptive labels. Perhaps a title as well.

```
plot(co2,col='red',lwd=2, xlab='Year',ylab='[CO2] (ppm)',
     main='Mauna Loa CO2 Concentration over Time')
```



What about some sample data from the CSHS-hydRology package.

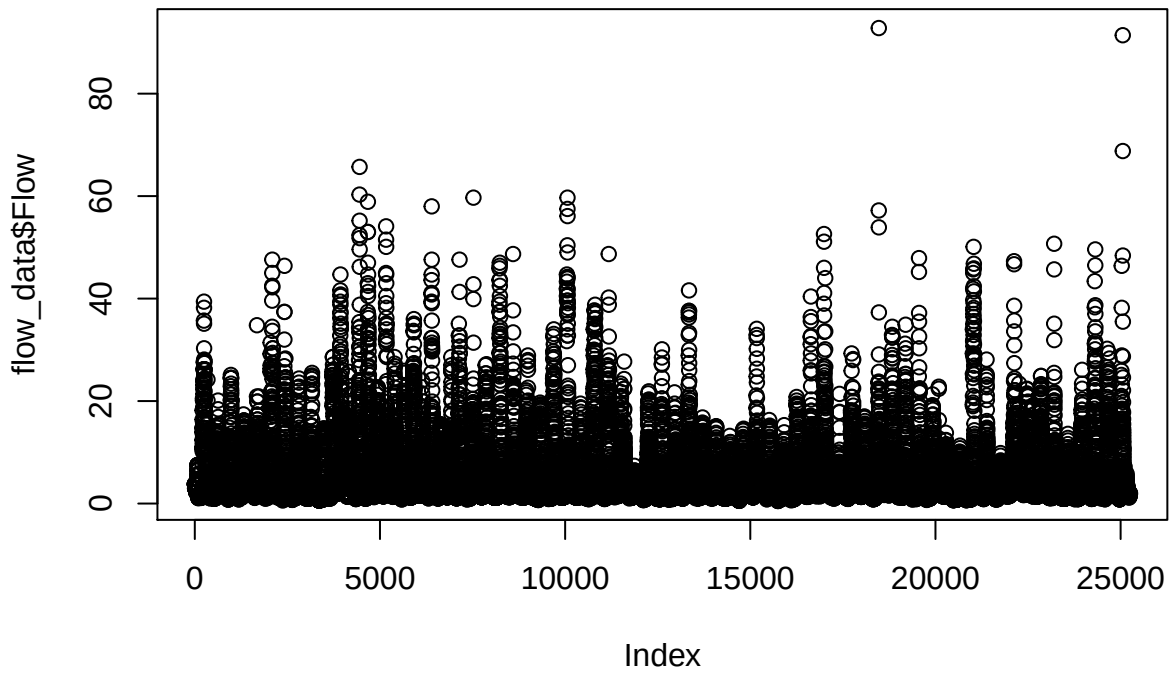
```
library(CSHShydRology)
data("W05AA008") # recalls data by this name from loaded packages

# preview and summarize the data to get a feel for it
head(W05AA008)
summary(W05AA008)

# see the documentation associated with this file
?W05AA008
```

We can see that this sample data set called **W05AA008** came from a Water Survey Canada gauge, and contains flow and level time series data. Let's try to make a plot with just the flow data, knowing all that we do about subsetting data and plotting.

```
library(CSHShydRology)
flow_data <- W05AA008[(W05AA008$PARAM == 1),] # all rows where we have flow data
# (i.e., PARAM is equal to 1)
head(flow_data) # preview our flow data
```

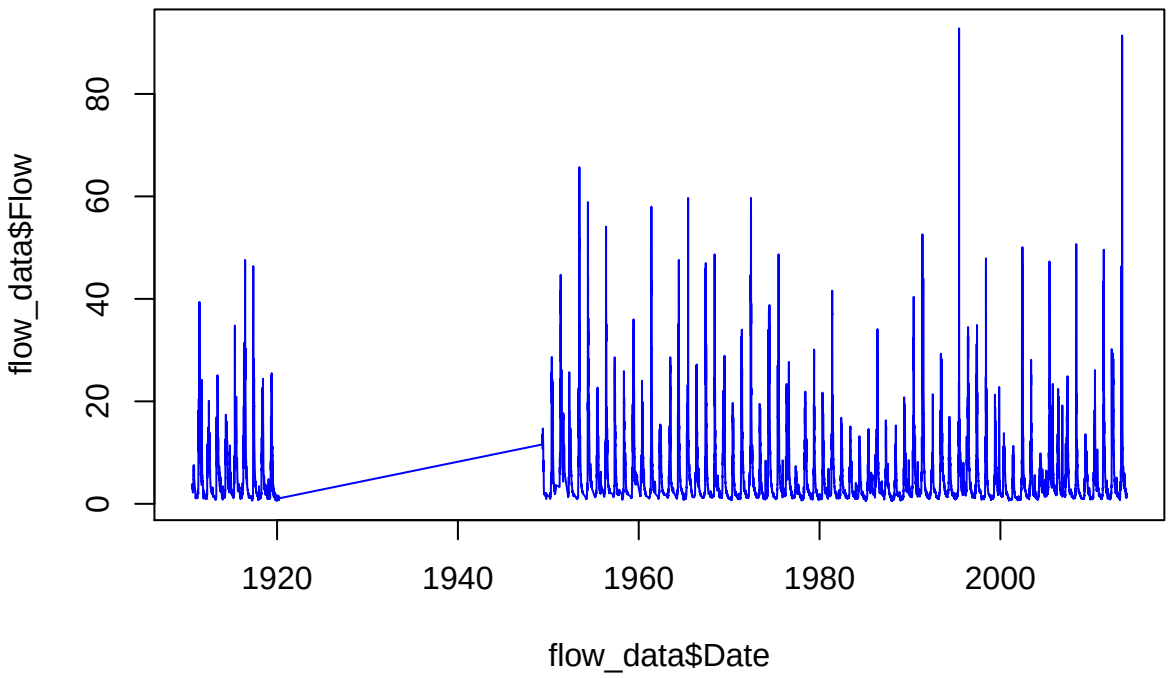
```
##        ID PARAM       Date Flow SYM
## 1 05AA008     1 1910-07-29 3.79
## 2 05AA008     1 1910-07-30 3.79
## 3 05AA008     1 1910-07-31 3.79
## 4 05AA008     1 1910-08-01 3.79
## 5 05AA008     1 1910-08-02 3.79
## 6 05AA008     1 1910-08-03 3.79
```

```
# make the plot
plot(flow_data$Flow)
```



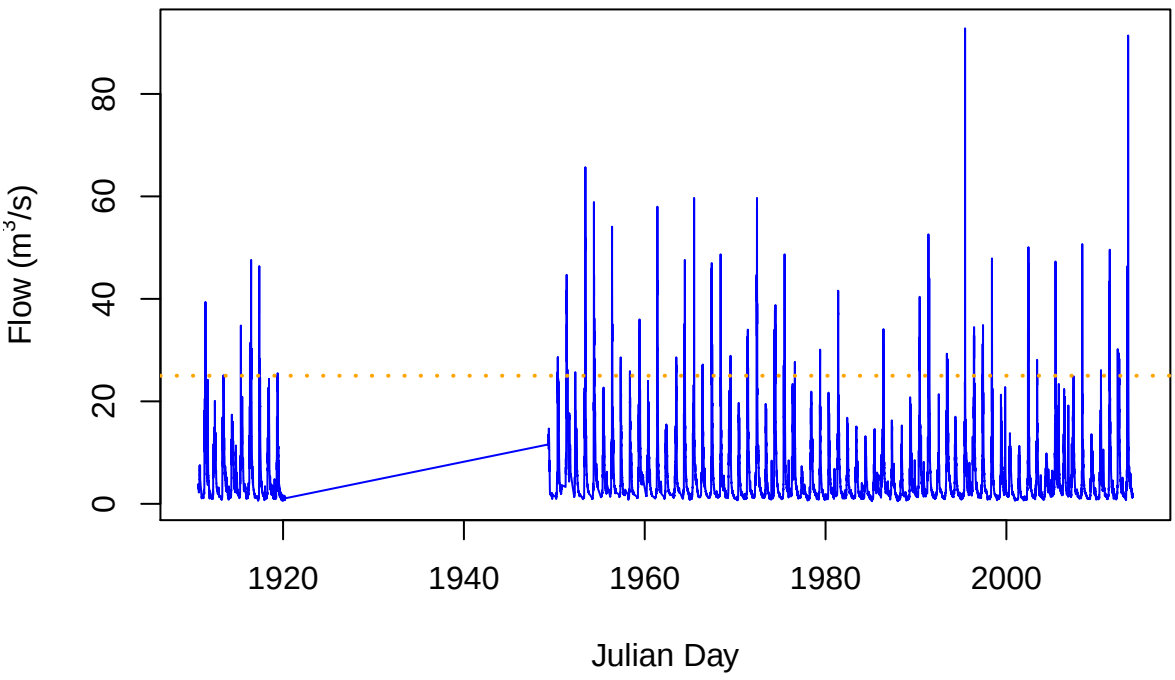
This looks like it could be flow data, but let's clean up the plot a litte bit. Since it is a time series, we may as well also plot it like one.

```
plot(flow_data$Date, flow_data$Flow,
     col='blue', type='l')
```



Now try to change the axis labels to something nice. Here is a little trick to get the cms label in units to work as a superscript.

```
plot(flow_data$Date, flow_data$Flow,
     col='blue', type='l',
     xlab="Julian Day",
     ylab=expression(paste("Flow (m"^"3","/s)")))

# add a threshold line
abline(h=25,col='orange',lty=3,lwd=2)
```



There are many ways to add special characters to plots. The "expression(paste())" formulation is one.

Feel free to check out the plotting options, try searching help on the plot function to bedazzle your plot, such as the **col** argument to change the colour of the plotted data.

Want to output your plot to an image file? Try this out.

```
# start a png file of the given name, all subsequent plotting is directed to the file
png("my_fabulous_plot.png")

plot(flow_data$Date, flow_data$Flow,
     col='blue', type='l',
     xlab="Julian Day",
     ylab=expression(paste("Flow (m"^"3/s)")))

# add a threshold line
abline(h=25,col='orange',lty=3,lwd=2)

dev.off() # release focus from the png file, return to RStudio

# check where the image was plotted to
getwd()

# see the files in the active folder
list.files()
```

You should see an image file (.png) in the active R folder (i.e. the one shown by the **getwd()** command). Ta-daaa!

Note that if you ever need to change the active directory you are in, you can use the **setwd** function. This may be needed if you want to work in a specific directory to conveniently load data during your session.

Note that if you open an R file with RStudio, the active directory will be automatically set to the one where the opened file is located.

As an aside, there is a hydograph plotting function in the CSHS-hydRology package. We will cover this in applications at the end of the document.

Basic Programming Stuff

We can't finish off a programming tutorial without some basic programming stuff. So here goes!

If statement

```
5 -> a
if (a < 2) {
  print("a is less than 2!")
} else {
  print("a is not less than 2!")
}
```

```
## [1] "a is not less than 2!"
```

Here the statement indicating that "a is not less than 2", since we defined 'a' with a value of 5. This same code would print a different statement (maybe) if we gave 'a' a different value.

Similarly, this script will check whether the number 4 is in our vector 'b', and print a statement (or not) accordingly.

```
b <- c(1,2,4,8)
if (4 %in% b) {
  print(sprintf("4 is contained in %s, which is c(%s)","b",paste(b,collapse = ", ")))
} else {
    print(sprintf("4 is not contained in %s, which is c(%s)","b",paste(b,collapse = ", ")))
}
```

```
## [1] "4 is contained in b, which is c(1, 2, 4, 8)"
```

```
b <- seq(1,9,2)
if (4 %in% b) {
  print(sprintf("4 is contained in %s, which is c(%s)","b",paste(b,collapse = ", ")))
} else {
    print(sprintf("4 is not contained in %s, which is c(%s)","b",paste(b,collapse = ", ")))
}
```

```
## [1] "4 is not contained in b, which is c(1, 3, 5, 7, 9)"
```

Loops

Loops of various forms are what give programming its power to take a task and repeat it loyally for everything we ask it to do. No more manual repetition! (or at least less of it).

These three types of loops are all setup to print the numbers 1 to 3 into the console, but with different syntax and different stopping conditions. See if you can figure out how they operate.

```
# for loops
for (i in 1:3) {
```

```
  print(i)
}
```

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

```
# while loop
i = 1
while(i < 4) {
  print(sprintf("%i is still less than 4",i))
  i = i +1
}
```

```
## [1] "1 is still less than 4"
## [1] "2 is still less than 4"
## [1] "3 is still less than 4"
```

```
# repeat loop
i = 1
repeat{
  print(i)
  if (i >= 3) {
    break
  }
  i = i + 1
}
```

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

Spark notes on loops:

- the **for** loop will repeat a task *for* the items it is given, then stop on its own
- the **while** loop will repeat a task *while* a given condition is true, and will break once that condition is no longer true
- the **repeat** loop will do just that until the loop is broken with a command such as **break**; note that the **break** command can be used to stop any of the loops above

A quick note on the speed of loops: loops in R are generally considered to be slow. Not in a way that is necessarily noticeable for most tasks, but there are faster and cleaner ways to run most things in R. For the hardcore coders, there is a general concensus that if you used a loop in R, then you have failed. For a lot of tasks this is not true (I still use loops for a lot of things) but it is good to get to know the quirks in R and avoid loops where possible.

For example, consider the following time tests on larger data sets.

```
dd <- as.numeric(seq(1,1e6)) # create a vector of 1 to 1million
ss <- 0 # sum to calculate

# get the sum of the vector
Sys.sleep(1) # pause for 1 second to let CPU settle
system.time(for (i in 1:length(dd)) {ss <- ss + dd[i]})
Sys.sleep(1)
system.time(sum(dd))

# take the sqrt of each element in the vector
```

```
Sys.sleep(1) # pause for 1 second to let CPU settle
system.time(for (i in 1:length(dd)) {dd[i] <- dd[i]^0.5})
dd <- as.numeric(seq(1,1e6)) # create a vector of 1 to 1million
Sys.sleep(1)
system.time(dd <- sqrt(dd))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
##    user  system elapsed
##       0       0       0
##    user  system elapsed
##    0.09    0.00    0.10
##    user  system elapsed
##    0.02    0.00    0.01
```

As you can see, using the built-in R functions is much faster than using a loop, especially when you have a lot of data to crunch (this difference becomes more and more noticeable the more data you have). The reason for this is that many of the built-in R functions use C language in the background, which is faster than using R itself (if that makes sense).

For example, try to look at the code for the *sqrt* and *sum* functions. It will show as "Primitive" functions, meaning it is internally implemented in non-R code, and runs faster than loops.

This may be more information than you need at this point, but the takeway is...avoid using loops where possible!

Applications in Hydrology and Water Resources with R

Let's take a look at some of the more useful packages and functions in R, as it relates to common tasks and usages of data.

Scraping Environment Canada Meteorological Data

Environment Canada maintains records of meteorological data at many stations across Canada. This data can be downloaded from the Historical Data page, however, you may have noticed that for hourly and daily downloads, you can only download one month at a time.

You can download many months one at a time and stitch them together in Excel, or you can use one of the existing R packages to scrape this data for you. Multiple packages exist to do this, including the rclimateca package on Github, below is an example with the weathercan package. We will download monthly data from the Port Weller station.

```
install.packages("weathercan")
```

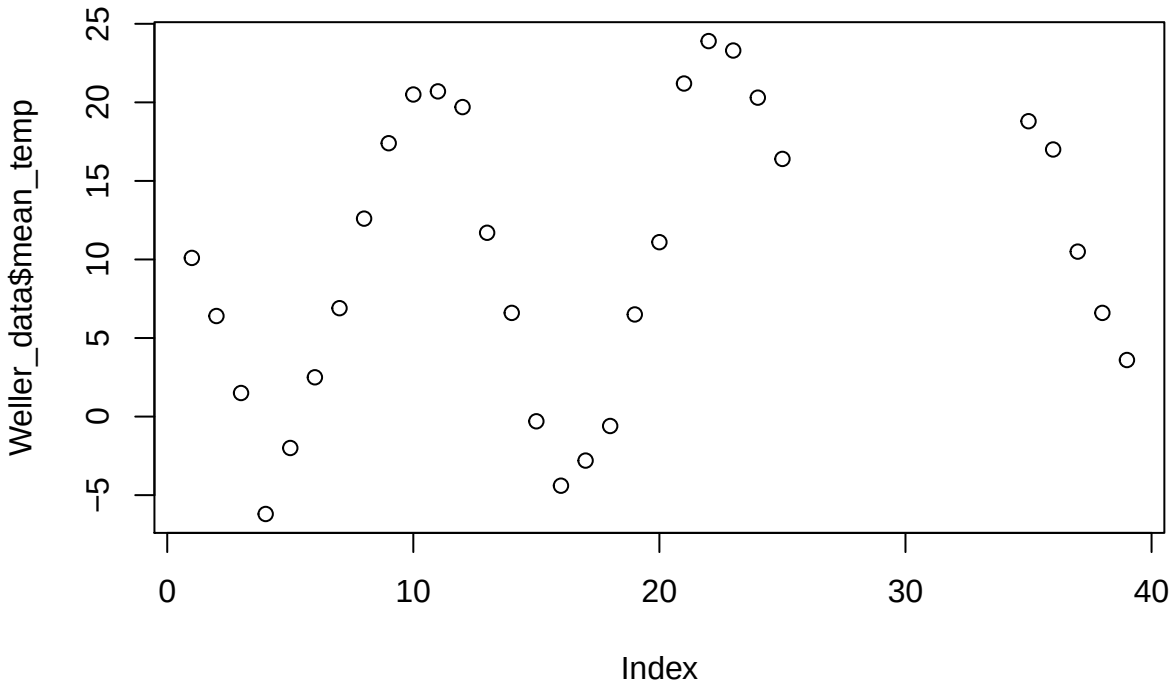
```
library(weathercan)
```

```
## Warning: package 'weathercan' was built under R version 3.4.4
```

```
weathercan::stations_search(name="Weller")
```

```
## # A tibble: 3 x 13
##   prov  station_name station_id climate_id WMO_id TC_id   lat   lon  elev
##   <fct> <chr>        <fct>      <fct>      <fct>  <fct> <dbl> <dbl> <dbl>
## 1 ON    PORT WELLER~ 7790       6136699    71432  WWZ    43.2 -79.2    79
## 2 ON    PORT WELLER~ 7790       6136699    71432  WWZ    43.2 -79.2    79
## 3 ON    PORT WELLER~ 7790       6136699    71432  WWZ    43.2 -79.2    79
## # ... with 4 more variables: tz <chr>, interval <chr>, start <int>,
## #   end <int>
```

```r
# scrape the Port Weller data with a monthly interval
Weller_data <- weathercan::weather_dl(station_ids = 7790,interval="month")
plot(Weller_data$mean_temp) # plot out monthly mean temperatures
```



Writing and Reading Data with .csv and .xlsx files

For those who require interaction with text, csv, and even Excel files, there are easy ways to read and write to many file types. Let's take a look at writing our Port Weller data to file for use with other programs.

```r
# Write out a tab delimited text file; the delimiter can be specified
write.table(x = Weller_data, file = "Port_Weller_data.txt",sep = "\t")

# csv file output, uses commas by default with the write.csv function
write.csv(Weller_data, file = "Port_Weller_data.csv")
```

You may notice, if you search for the help on **write.csv**, that **write.csv** is actually a wrapper function on the more generic write.table. If you use **write.table** with a comma delimiter and give the file a csv extension, this is by definition a comma-separated values file. The **write.csv** function just saves you a couple steps in writing the command.

In order to read a file, the commands are quite similar. We will not cover this here, but look at the help for the **read.table** and **read.csv** functions.

In order to interact with Excel files, there are a number of packages which enable you to read and write from Excel files. Take a look at the **xlsx** package as an example.

```r
install.packages("xlsx")
```

```
library(xlsx)
```

```
## Warning: package 'xlsx' was built under R version 3.4.4
```

```
xlsx::write.xlsx(x=Weller_data, file="Port_Weller_Data.xlsx",sheetName = "monthly_data")
```

Scraping Water Survey of Canada Data

The Water Survey of Canada records flow and water levels at many gauge stations throughout Canada. This data is available online and may be downloaded direclty from the Historical Data Page. However, there is a useful package to download this data directly into R without the need to open the browser and download multiple files. This can be done with the tidyhydat package, developed by Sam Albers.

```
install.packages("tidyhydat") # package is available on CRAN direct
library(tidyhydat)
# This will download the HYDAT database, required to run the package.
# This may take a while!
download_hydat()
```

The *download_hydat()* command downloads the HYDAT database, which is also what is used in the ECCC Data Explorer tool when searching for stations. Once this database is downloaded, the data for individual stations can be queried and easily downloaded.

Let's take a look at data from the Grand River; we will download data for the Grand River at Galt (02GA003) and the Grand River at Brantford (02GB001). By default, the **hy_daily_flows** function will download the data for the specified stations for the entire period available; the function can also specify particular dates to download.

```
library(tidyhydat)
```

```
## Warning: package 'tidyhydat' was built under R version 3.4.4
```

```
# Grand River at Galt daily flows
galt <- hy_daily_flows(station_number = "02GA003")
brantford <- hy_daily_flows(station_number = "02GB001")
summary(galt)
```

```
##  STATION_NUMBER          Date              Parameter
##  Length:38535       Min.   :1913-07-01   Length:38535
##  Class :character   1st Qu.:1939-11-15   Class :character
##  Mode  :character   Median :1966-04-01   Mode  :character
##                     Mean   :1966-04-01
##                     3rd Qu.:1992-08-15
##                     Max.   :2018-12-31
##
##      Value             Symbol
##  Min.   :   0.736   Length:38535
##  1st Qu.:  12.700   Class :character
##  Median :  19.500   Mode  :character
##  Mean   :  37.376
##  3rd Qu.:  37.100
##  Max.   :1140.000
##  NA's   :20
```

Date and Time Series Handling

Handling of date and time data is certainly not unique to water resources, but it is a common task within any environmental discipline where data logging is common. There are many packages available to streamline handling and manipulating time series data, and you can also do a lot with the base R functions without additional packages.

The handling of time series data may be the topic of a future webinar, but here are a few snippets of handling time series data which you may find useful. Let's use the Port Weller csv file created from earlier in this document.

```
mydata <- read.csv("Port_Weller_data.csv",header=T,stringsAsFactors = F)
class(mydata$date)
```

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

You will notice that when the data is read in, the data is read in as a character (a text string) rather than a date. We need to convert this to let R know that it is a date, not just random characters.

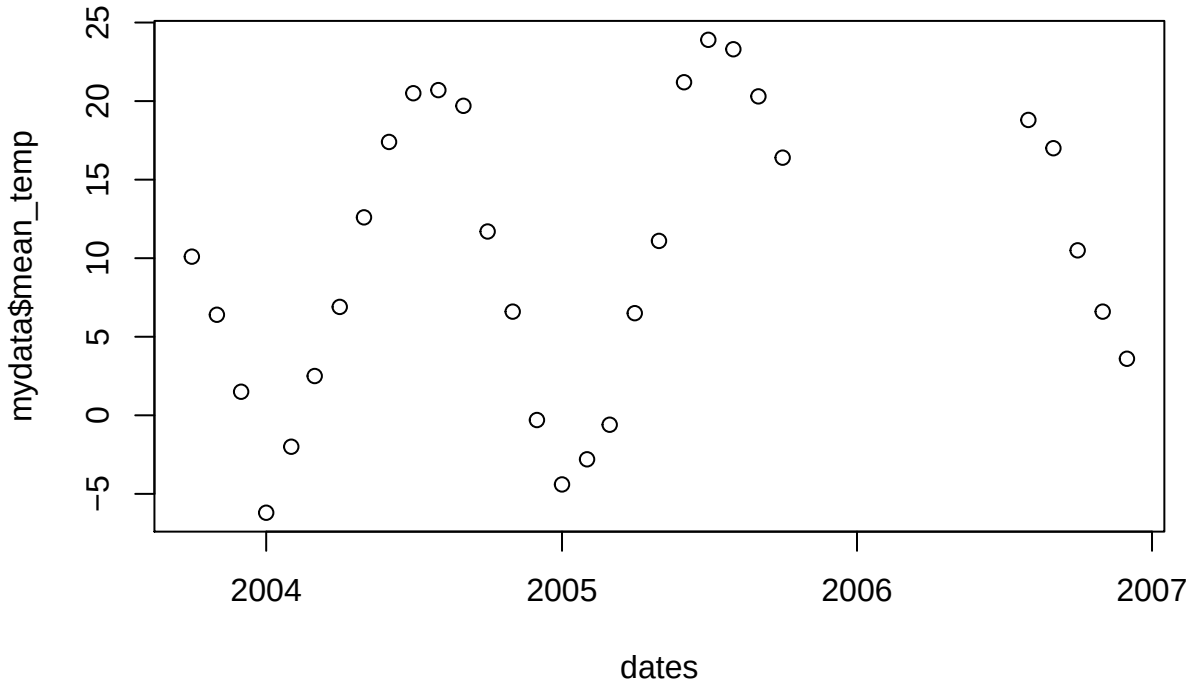
```
dates <- as.Date(mydata$date,format = "%Y-%m-%d")
class(dates)
```

```
## [1] "Date"
```

```
head(dates) # looks the same, but is not a character vector any longer
```

```
## [1] "2003-10-01" "2003-11-01" "2003-12-01" "2004-01-01" "2004-02-01"
## [6] "2004-03-01"
```

```
plot(dates,mydata$mean_temp)
```



You will notice that now when we provide the date values as a 'Date' object to the plot on the x-axis, R knows how to handle the dates and automatically plots the year, based on the resolution. We could adjust this to plot the year-month, etc.

The format of the data we read in follows a year-month-day format separated with hyphens, so this is the format we tell R to look for. The same could be done with backslashes, a year-day-month format, a two-cahracter year representation ("99-11-05"), etc. Similar syntax exists for subdaily time series, where an "H:m:s" format can also be specified. There are multiple functions to convert to date/time data, but this was a quick example.

A useful package for handling time series is the **lubridate** package, which helps to streamline much of the conversion with simpler functions than some of the base R ones. There are also useful functions for extracting the year and months out of a date object, getting the yday (1-365) for a given date, etc.

```
install.packages("lubridate")
```

```
lubridate::month(dates)
```

```
##  [1] 10 11 12  1  2  3  4  5  6  7  8  9 10 11 12  1  2  3  4  5  6  7  8
## [24]  9 10 11 12  1  2  3  4  5  6  7  8  9 10 11 12
```

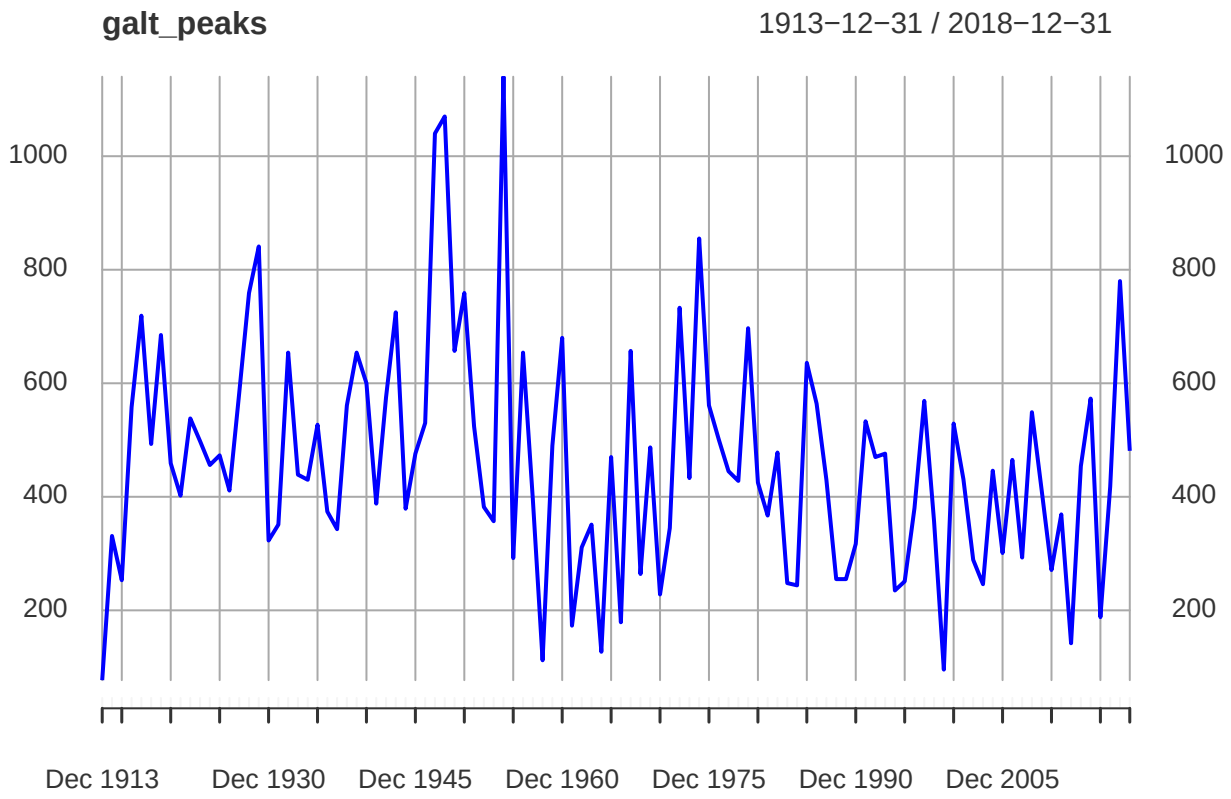
```
lubridate::yday(dates)
```

```
##  [1] 274 305 335   1  32  61  92 122 153 183 214 245 275 306 336   1  32
## [18]  60  91 121 152 182 213 244 274 305 335   1  32  60  91 121 152 182
## [35] 213 244 274 305 335
```

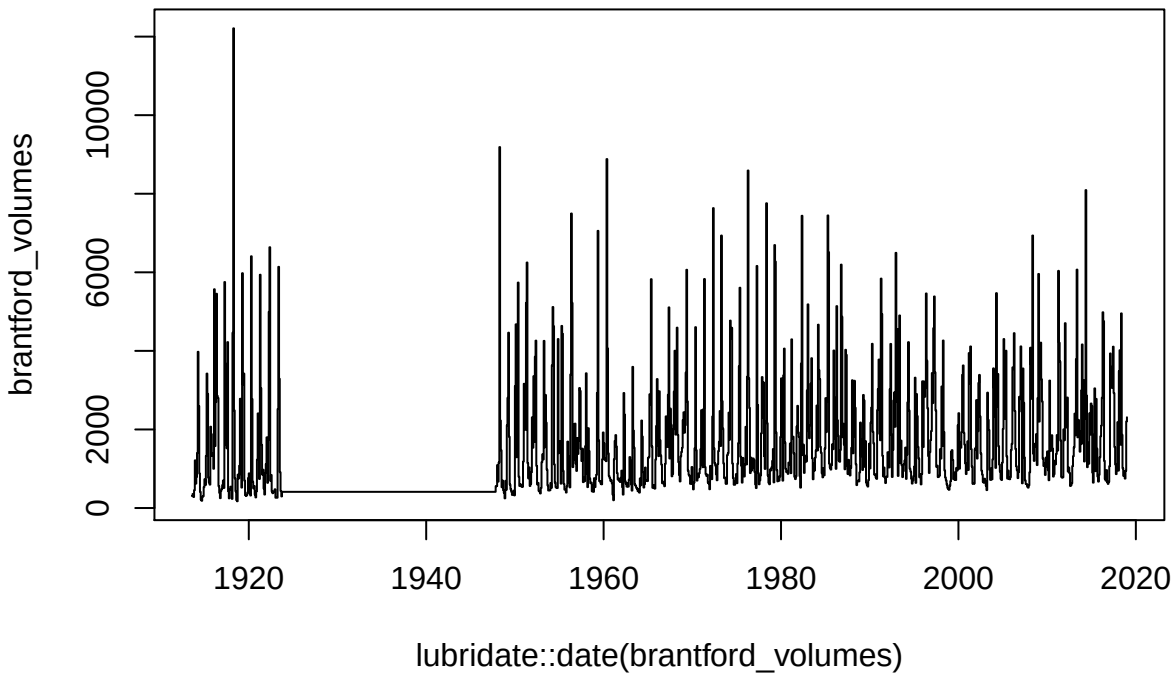
Another useful package, particularly on analyzing and applying operations to time series, is the **xts** package. This one has a lot of powerful functionality to do things with time series data once you have it in the xts format. Let's work with the flow data downloaded from the tidyhydat package to look at hydrograph time series data.

```
install.packages("xts")
```

```
# first, convert our time series data to the xts format
galt_xts <- xts(galt$Value, order.by=galt$Date)
brantford_xts <- xts(brantford$Value, order.by=brantford$Date)

# Now we can access the xts functions to apply functions at different intervals.
# For example, extracting annual peak flows
galt_peaks <- apply.yearly(galt_xts,max,na.rm=T)
plot(galt_peaks,ylab="Flow (cms)",col='blue')
```




```
# default plot for xts objects, date is handled automatically on x-axis

# Or look at monthly flow volumes
brantford_volumes <- apply.monthly(brantford_xts,sum,na.rm=T)
# Instead we can pull out the date ourselves and plot with regular base plots
plot(lubridate::date(brantford_volumes),brantford_volumes,type='s')
```

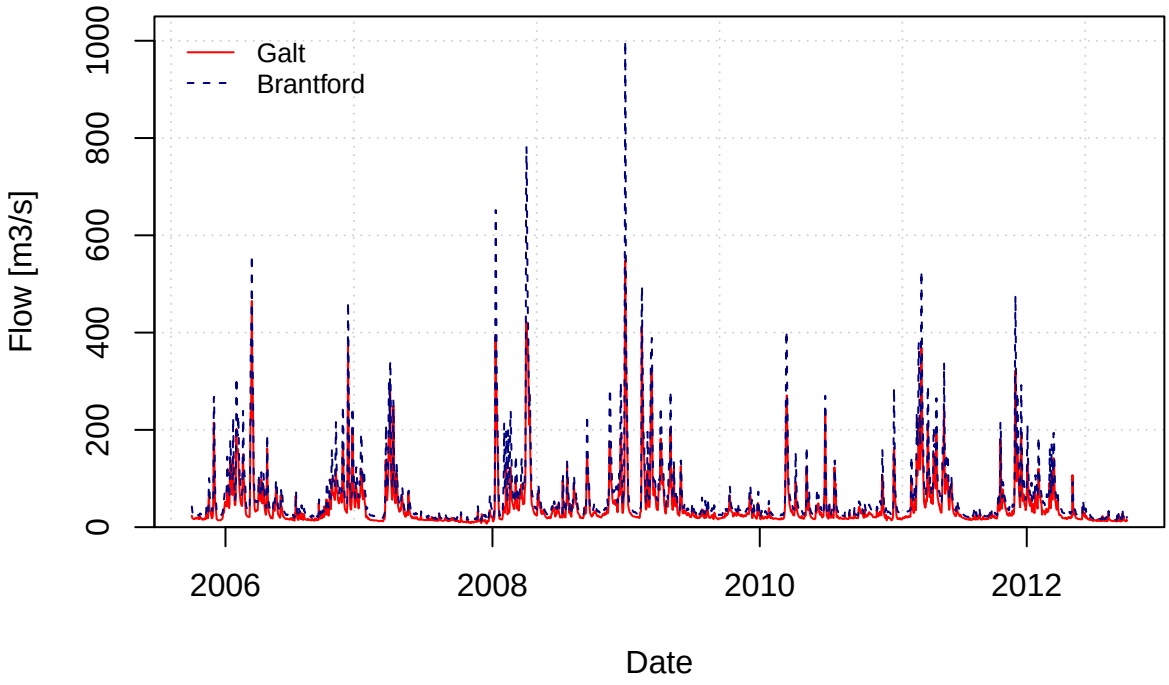


Notice that the plotting with xts objects has its own default look, where R knows how to find the date for an xts object and plots it without requiring us to specify the x-axis with date values. However, we can do this if we wish to customize the look of the plot.

CSHS-hydRology hydrograph plot

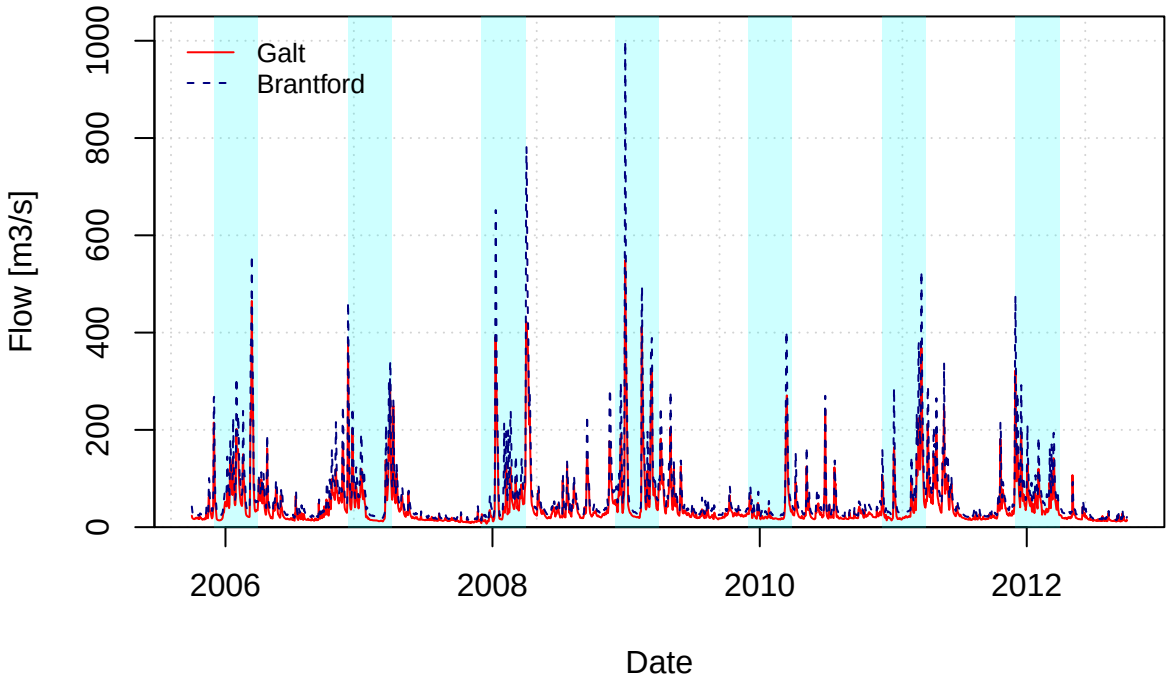
There are a lot of useful functions in the hydRology package, but a useful one for a basic hydrograph plot is the **hydrograph_plot** function. Making a simple plot is not too hard, but still has a few steps. This function simplifies the process.

```
prd <- "2005-10-01/2012-10-01" # common period of interest to use

# build a non-xts data frame for the hydrograph functions from xts data
df <- data.frame("Date"=seq.Date(from=as.Date("2005-10-01"),to=as.Date("2012-10-01"),by=1),
                 as.numeric(galt_xts[prd,]),
                 as.numeric(brantford_xts[prd,]))

# plot the hydrograph
hydrograph_plot(df,flow_labels = c("Galt","Brantford"))
```



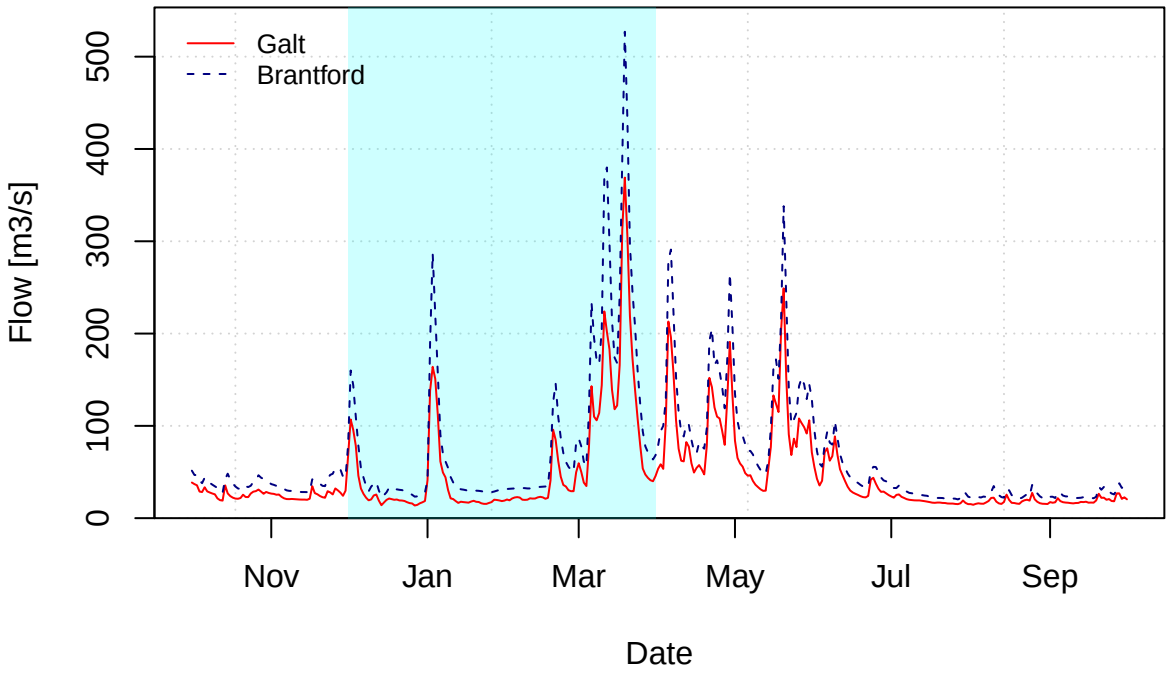

```
## [1] TRUE
```

```
# plot with winter shading for visualization
hydrograph_plot(df,flow_labels = c("Galt","Brantford"), winter_shading = TRUE)
```




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

```
# change the plotting period on the fly for just 2010/2011 water year
hydrograph_plot(df,flow_labels = c("Galt","Brantford"), winter_shading = TRUE,
                prd="2010-10-01/2011-10-01")
```


Galt
Brantford
Flow [m3/s]
0
100
200
300
400
500
Nov
Jan
Mar
May
Jul
Sep
Date

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