# R Syntax Examples

Nathan Young

## R Nuts and Bolts

### Entering Input

We type expressions into the R prompt. The <- symbol is the assignment operator

```
x <- 5 # Assignment of value to variable
print(x) # Explicit printing
```

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

```
x # auto-printing
```

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

```
y <- 10:20
```

When [1] is printed it indicates that x is a vector and that 5 is the first element The : operator is used to create integer sequences

### R Objects

R has five basic classes of objects - Character - Numeric (real numbers) - Integer - Complex - Logical (True/False)

The vector is the most basic object type, initialized with vector() function.
A vector can only contain objects of the same class A *list* is represented as a vector but can contain objects of different classes.

### Numbers

Numbers are generally treated as numeric objects (such as double precision real numbers). Meaning, even if you see a '1' it is represented behind the scene as a numeric object like '1.00'. If you explicitly want an integer, use the L suffix. Entering '1' gives you a numeric object, while entering '1L' gives an integer object. Inf represents infinity. NaN represents an undefined value such as 0/0

### Attributes

R Objects can have many attributes (like metadata) to help describe the object. * Names, dimnames * dimensions (matrices, arrays) * class (integer, numeric) * length * other user-defined attributes/metadata

Attributes can be accessed using attributes() function.

## Creating Vectors

The c() function can be used to create vectors of objects by concatenating things together.

```
x <- c(0.5, 0.6)       # numeric
x <- c(TRUE, FALSE)    # logical
x <- c(T, F)           # logical
x <- c("a", "b", "c")  # character
x <- 9:29              # integer
x <- c(1+0i, 2+4i)     # complex
x <- vector("numeric", length = 10) # initialize vector
```

## Mixing Objects

Sometimes we mix objects (on accident or on purpose).
When different objects are mixed in a vector, they are *coerced* to be the same class. Sometimes this works the way you expect others, not.

## Explicit Coercion

You can explicitly coerce between classes using the as.* function

```
x <- 0:6
class(x)
```

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

```
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R doesn't know what to do and will return NA.

## Matrices

Vectors with a *dimension* attribute. The dimension attribute is an integer vector of length 2

```
m <- matrix (nrow = 2, ncol = 3)
dim(m)
```

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

```
attributes(m)
```

```
## $dim
## [1] 2 3
```

Matrics are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
m <- matrix(1:6,nrow = 2, ncol = 3)
m
```

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

Can also be created directly from vectors by adding a dimension attribute.

```
m <- 1:10
m
```

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

```
dim(m) <- c(2,5)
m
```

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

Can be created with *column-binding* or *row-binding*

```
x <- 1:3
y <- 10:13
cbind(x,y)
```

```
## Warning in cbind(x, y): number of rows of result is not a multiple of vector
## length (arg 1)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
## [4,] 1 13
```

```
rbind(x,y)
```

```
## Warning in rbind(x, y): number of columns of result is not a multiple of vector
## length (arg 1)
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    1
## y   10   11   12   13
```

## Lists

Special vectors that contain different elements of different classes.
Can be explicitly created with the list() function.

```
x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Can also create an empty list of length with the vector() function.

```
x <- vector("list", length = 5)
```

## Factors

Used to represent categorical data that is unordered or ordered. Akin to integer vector where each integer has a label. Important for statical modeling and treated special in functions like lm() and glm().
Using factors with labels is better than using integers because factors are self-describing. Factor with values "Male" and "Female" is better than variable with values 1 and 2. Create with factor() function

```
x <- factor(c("yes", "yes", "no", "yes", "yes"))
x
```

```
## [1] yes yes no  yes yes
## Levels: no yes
```

```
table(x)
```

```
## x
##  no yes
##   1   4
```

```
unclass(x)
```

```
## [1] 2 2 1 2 2
## attr(,"levels")
## [1] "no"  "yes"
```

Often, factors are automatically created when reading data with function like data.table().
Order of levels can be set using levels argument to factor(). Often important in linear modeling with first level as baseline.

## Missing Values

Denoted by NA or NaN for undefined mathematical operations - is.na() is used to test for NA - is.nan() is used to test for NaN - NA values have a class also (integer NA, character NA, etc) - A NaN value is also NA

```
# create a vector with NAs in it
x <- c(1,2,NA,10,NaN,3)
# Return logical vector indicating which elements are NA
is.na(x)
```

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

```
# Return logical vector indicating which elements are NaN
is.nan(x)
```

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

## Data Frames

Used to store tabular data in R. Important in many statistical modeling. dplyr has optimized set of functions to work with data frames.
Represented as special type of list where everyelement of the list has to have the same length. Each element can be thought of as a column and the length of each element of the list is the number of rows.
Unlike matrices, can store different classes of objects in ach column.
In addition to column names, DF have attribute called row.names to indicate informration about each row.
Usually created by reading a dataset using read.table() or read.csv(). Can be created with data.frame() or converted from other objects.
Can be converted to matrix with data.matrix().

```
x <- data.frame(foo = 1:4, bar = c(TRUE, TRUE, FALSE, FALSE))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

## Names

R objects can have names, helpful for readable code.

```r
x <- 1:3
names(x)
```

```
## NULL
```

```r
names(x) <- c("New York", "Seattle", "Los Angeles")
x
```

```
##    New York     Seattle Los Angeles
##           1           2           3
```

```r
names(x)
```

```
## [1] "New York"    "Seattle"     "Los Angeles"
```

```r
# Lists can also have names
x <- list("Los Angeles" = 1, Boston = 2, London = 3)
x
```

```
## $`Los Angeles`
## [1] 1
##
## $Boston
## [1] 2
##
## $London
## [1] 3
```

```r
names(x)
```

```
## [1] "Los Angeles" "Boston"      "London"
```

```r
# Matrices can have both row and column names
m <- matrix(1:4, nrow = 2, ncol = 2)
dimnames(m) <- list(c("a","b"),c("c","d"))
m
```

```
##   c d
## a 1 3
## b 2 4
```

```r
# Column names and row names can be specified separately
colnames(m) <- c("h","f")
rownames(m) <- c("x","z")
m
```

```
##   h f
## x 1 3
## z 2 4
```

Note that for dataframes, there is a separate function for setting row names, the row.names(). DF do not have column names, just names. So you use the names() function.

# Getting Data In and Out of R

## Reading and Writing Data

There are a few principal functions - read.table, read.csv for reading tabluar data - readLines for reading lines of a text file - source, for reading in R code files (inverse of dump) - dget, for reading in R code files (inverse of dput) - load, for reading in saved workspaces - unserialize, for reading simple R objects in binary form.

Many packages exist for other datasets

Analogous functions for writing to files - write.table for writing tabular to text files or connections - write-Lines, for writing character data line-by-line to file or conn - dump, for dumping a textual representation of multiple R objects - dput, for outputting a textual representation of an R object - save, for saving an arbitrary number of R objects in binary format to a file - serialize, for converting an R object into a binary format for outputting to a connection.

## Reading Data Files with read.table()

Commonly used. Help file is worth reading. Arguments: - file, name of file, or connection - header, logical indicating if file has a header - sep, a string indicating how columns are separated - colClasses, character vector indicating the class of each column - nrows, number of rows. Default reads whole file - comment.char, chacter string indicating comment character. - skip, number of lines to skip from beginning - stringsAsFactors, should character variables be coded as factors? defaults to True.

For small to moderate size, you can usually call read.table

```
#data <- read.table("foo.txt")
```

- In this case, R will automatically skip lines that begin with #
- figure out how many rows there are and how much mem is needed.
- figure out type of variable in each column.

Telling R these things directly makes R faster and more efficient.

## Reading in larger data sets with read.table

With larger data sets, make life easier and prevent R from choking by: - Read the help page for read.table - Make a rough calculation of the memory required to store your dataset. If dataset is larger than RAM, stop. - set comment.char = " " if no commented lines in file - Use colClasses argument. Specifing this can make it run **much** faster, often twice as fast.

```
#initial <- read.table("datatable.txt",nrows=100)
#classes <- sapply(initial,class)
#tabAll <- read.table("datatable.txt",colClasses=classes)
```

- set nrows. Doesn't make faster but helps with memory.

Also useful to know about your system. - How much mem is available? - Can you close other applications? - Are there other users logged in? - What OS are you using? Some limit amount of memory.

### Calculating Memory Requirements for R Objects

Because R stores all objects in physical memory, important to keep limits in mind. Especially when reading in a new dataset to R. BOTE are easy for it.
Suppose data frame with 1,500,000 rows and 120 columns all of which are numeric. How much memory to store numeric data? 1,500,000 x 120 x 8 bytes/numeric = 1,440,000,000 bytes = 1,400,000,000 / 2^20 bytes/MB = 1,373.29 MB = 1.34 GB.

## Using the readr Package

Recently developed to deal with reading large flat files quickly, replacement for read.table() and read.csv(). Ananogous in readr are read_table() and read_csv().
For the most part, can use these instead.

```
library(readr)
#teams <- read_csv("data/team_standings.csv")
#teams
```

By default, will open and read lint-by-line. Will also, read first few rows of table to figure out type of column. Can instead specify type of each column with col_types argument. Good idea in general.
col_types accepts a compact representation. "cc" says first and second columns are characters.
read_csv also reads compressed files automatically, no need to decompress.
Can specify column type in detailed fashion using various col_* functions.

## Using Textual and Binary Formats for Storing Data

Data can be stored in a variety of ways including strutured forms like CSV. Intermediate format that is textual, but not as simple as CSV.
One can create a more descriptive representation of an R object by using the dput() or dump() functions; useful because resulting textual format is editable and recoverable in case of corruption. Preserve metadata (sacrificing readability) so user doesn't have to specify it again.
Work much better with VC programs to track meaningful changes.
Downsides include not space inefficiency and partial readibility.

### Using dput() and dump()

One way to pass data around is deparsing the R object with dput() and reading it back in with dget().

```
# Create a data frame
y <- data.frame(a = 1, b = "a")
# Print 'dput' output to console
dput(y)
```

```
## structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

Notice that the dput() output is in the form of R code and that it preserves meta data like class, row names, and column names.
Output can also be saved directly to a file

```
# Send 'dput' output to a file
dput(y, file = "y.R")
# Read in 'dput' output from a file
new.y <-dget("y.R")
new.y
```

```
##   a b
## 1 1 a
```

Multiple objects can be deparsed at once using the dump function and read back in using source

```
x <- "foo"
y <- data.frame(a = 1L, b = "a")
dump(c("x","y"),file = "data.R")
rm(x,y)
# Inverse of dump is source
source("data.R")
str(y)
```

```
## 'data.frame':    1 obs. of  2 variables:
##  $ a: int 1
##  $ b: chr "a"
```

```
x
```

```
## [1] "foo"
```

## Binary Formats

The complement to the textual format is the binary format. Use for efficiency purposes or if no useful way to represent data with text. Also, can lose precision on numeric data when converting to and from textual format.
Convert R object to binary: save(), save.image(), serialize().
Individual objects with save() function

```
a <- data.frame(x = rnorm(100), y = runif(100))
b <- c(3, 4.4, 1/3)
# Save 'a' and 'b' to a file
save(a, b, file = "mydata.rda")
# Load 'a' and 'b' into workspace
load("mydata.rda")
```

Save lots of objects with save.image()

```
# Save everything
save.image(file = "mydata.RData")
# Load all objects in this file
load("mydata.RData")
```

Example used .rda for save and .RData for save.image. Can use other formats but these are common and accessible.
serialize() converts individual R objects to binary format that can be communicated across an arbitrary connection (file or network).
Coded as a raw vector in hexadecimal.

```
x <- list(1, 2, 3)
serialize(x,NULL)
```

```
##  [1] 58 0a 00 00 00 03 00 04 02 02 00 03 05 00 00 00 00 05 55 54 46 2d 38 00 00
## [26] 00 13 00 00 00 03 00 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00
## [51] 0e 00 00 00 01 40 00 00 00 00 00 00 00 00 00 00 0e 00 00 00 01 40 08 00 00
## [76] 00 00 00 00
```

Can be sent to a file, but better of using save for that.
Benefit to serialize: perfectly represent R object in an exportable format, without losing precision or metadata.

## Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files or more exotic things. - file, opens connection to file - gzfile, opens connection to file compressed with gzip - bzfile, opens connection to file compressed with bzip2 - url, opens connection to webpage.

Think of connections as translator that lets you talk to objects outside of R.
## File Connections Connections to file using file()

```
str(file)
```

```
## function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"),
##     raw = FALSE, method = getOption("url.method", "default"))
```

Arguments:

- description, name of file
- open, mode to open file
  - 'r' read mode
  - 'w' write mote (and initialize new file)
  - 'a' appending
  - 'rb', 'wb', 'ab' readin writing appending in binary

Most of connection is dealt with in background and we don't deal with it.

```
# Create connection to 'foo.txt'
con <- file("foo.txt")
# Open connetion in read mode
open(con, "r")
# Read from connection
data <- read.csv(con)
# Close connection
```

```
close(con)

# this is all the same as:
data <- read.csv("foo.txt")
```

In the background, read.csv() opens a connection, reads, and closes the connection.

## Reading Lines of a Text File

Lines of text file can be read line by line with readLines(). Useful for unstructured and nonstandard data.

```
# Open connection to gz-compressed file
# con <- gzfile("words.gz")
# x <- readLines(con,10)
# x
```

For more structured text like CSV, other functions like read.csv() and read.table() exits.
In above, gzfile() is used to create connections to compressed data with gzip algorithm. Lets you not unzip files.
Complementary function writeLines() to write character vector element wise to each line.

### Reading from a URL Connection

readLines() is useful for reading in lines of webpages that are text files stored on a server. url() function navigates communication between computer and web server.

```
# Open a url connection for reading
con <- url("https://www.jhu.edu","r")

# Read the webpage
x <- readLines(con)

# Print first few lines
head(x)
```

```
## [1] "<!doctype html>"                    ""
## [3] "<html class=\"no-js\" lang=\"en\">" "  <head>"
## [5] "    <script>"                       "    dataLayer = [];"
```

Reading a simple page is sometimes useful, more commonly used to read data stored on web servers.
Using URL connections can be useful for producing a reproducible analysis, because code documents where the data cam from and how it was obtained. Preferable to opening a webpage and downloading dataset by hand. If server is changed, can break code.

## Subsetting R Objects

Three operators to extract subsets of R Objects

- The "[]" operator always returns an object of the same class as the original, can be used to select multiple elements of object
- [[]] operator is used to extract elements of a list or data frame. Used to extract single element and class of returned object will not necessarily be a list or data frame.
- $ operator is used to extract elements of a list by literal name.

### Subsetting a Vector

Vectors are basic objects and are subsetted using the [] operator.

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]
```

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

```
x[2]
```

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

```
# Can be used to extract multiple elements by passing an inteer sequence.
x[1:4]
```

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

```
# Sequence doesn't need to be in order
x[c(1, 3, 4)]
```

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

```
# Can also pass logical sequence. For example, elements of x that come alphabetically after letter a
u <- x > "a"
x[u]
```

```
## [1] "b" "c" "c" "d"
```

```
# More compactly:
x[x > "a"]
```

```
## [1] "b" "c" "c" "d"
```

## Subsetting a Matrix

Subsetted in usual way with $(i, j)$ indices. Row x column

```
x <- matrix(1:6, 2, 3)
x
```

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

```
# access elements using indices
x[1,2]
```

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

```
x[2,1]
```

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

```
# Missing index indicates entire row/column
x[1,]
```

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

```
x[,2]
```

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

### Dropping Matrix Dimensions

By default, when you index a single element a vector length 1 is returned. Can turn this off.

```
x <- matrix(1:6, 2, 3)
x[1,2]
x[1, 2, drop = FALSE]
# Similar use to keep matrix dimensions for extracting entire row.
x[1,]
x[1, , drop = FALSE]
```

### Be careful of R's automatic dropping of dimensions

## Subsetting Lists

```
x <- list(foo = 1:4, bar = 0.6)
x
```

```
## $foo
## [1] 1 2 3 4
##
## $bar
## [1] 0.6
```

```
## The [[]] operator can be used to extract a single element from a list
x[[1]]
```

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

```
## Can also use named indices or $ operator to extract by name
x[["bar"]]
```

```
## [1] 0.6
```

```
x$bar
```

```
## [1] 0.6
```

```
# Don't need quotes on $ operator.
# [[ can be used with computed indices, $ operator can only be used with names
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"

# Computed index for "foo"
x[[name]]
```

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

```
# element name doesn't exist!
x$name
```

```
## NULL
```

### Subsetting Nested Elements of a List

The [[]] operator can take an integer sequence

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
# Get 3rd element of the first element
x[[c(1,3)]]
```

```
## [1] 14
```

```
# Alternative
x[[1]][[3]]
```

```
## [1] 14
```

```
# first element of second element
x[[c(2,1)]]
```

```
## [1] 3.14
```

### Extracting Multiple Elements of a List

The [] operator can extract multiples

```r
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1,3)]
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

Note that x[c(1,3)] is not the same as x[[c(1,3)]]!

# Dates and Times

R has a special representation for date and time with. Date class for dates and POSIXct or POSIXlt for times. Date is stored as number of days since 1970-01-01 while time is number osf seconds since 1970-01-01.

## Dates in R

Dates are represented by Date class and can be coerced from a character string with as.Date().

```r
# Coerce a date object from character
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```r
# You can see internal representation of Date object by using the unclass() function.
unclass(x)
```

```
## [1] 0
```

```r
unclass(as.Date("1970-01-02"))
```

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

## Times in R

Times are represented by POSIXct or POSIXlt. POSIXct is just a large integer underneath and useful for storing in data frame. POSIXlt is a list underneath and stores extra info like day of week, year, month, day of month. Useful if you need extra info.

Generic Functions

- weekdays: gives day of week
- months: gives the month dame
- quarters: gives the quarter number "Q1" etc.

Times can be coerced from character string using as.POSIXlt or as.POSIXct

```r
x <- Sys.time()
x
```

```
## [1] "2022-12-11 15:57:06 EST"
```

```r
class(x)
```

```
## [1] "POSIXct" "POSIXt"
```

```r
# POSIXlt has useful metadata
p <- as.POSIXlt(x)
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```r
p$wday
```

```
## [1] 0
```

```r
# can also use POSIXct format
x <- Sys.time()
x # already in POSIXct format
```

```
## [1] "2022-12-11 15:57:06 EST"
```

```r
unclass(x) # Internal representation
```

```
## [1] 1670792226
```

```r
# x$sec #Won't work!
p <- as.POSIXlt(x)
p$sec
```

```
## [1] 6.132686
```

```r
# strptime() for dates in different format. Takes character vector and converts them to POSIXlt object.
datestring <- c("January 1, 2012 10:40", "December 9, 2011 9:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-01 10:40:00 EST" "2011-12-09 09:10:00 EST"
```

```r
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

The % symbols are for formatting strings for dates and times. Probably not worth memorizing. Check ?strptime for details.

### Operations on Dates and Times

Can use +, - and comparisions (==, <=, etc)

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
# x - y doesn't work!
x <- as.POSIXlt(x)
x - y
```

```
## Time difference of 356.3095 days
```

A nice feature of this class is it keeps track of leap years, daylight savings, time zones.

```
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
x - y
```

```
## Time difference of 2 days
```

```
# Timezone
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz="GMT")
y - x
```

```
## Time difference of 1 hours
```

## Managing Data Frames with the dplyr package

### Data Frames

DF are a key structure. Basic structure is one observation per row and each column represents a variable, measure, feature, or characteristic. Extra formats downloadable from CRAN that give better implementation for things like large datasets. Tools for dealing with them include subset(), [], and $ to extract subsets. Other operations with base R like filtering, reordering, and collapsing can be tedious. dplyr helps mitigate and optimize these challenges.

### The dplyr Package

Doesn't introduce anything new, but it *greatly* simplifies existing functionality. It provides a "grammar" for data manipulation and operating on data frames. Helps communicate what you are doing to a data frame. Also, dplyr is very fast.

### dplyr Grammar

key "verbs":

- select: return a subset of the columns of a data frame, using flexible notation

- filter: extract a subset of rows from data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in data frame
- mutate: add new variables/columns or transform existing variables.
- summarize/summarise: generate summary table of statistics of different variables in data frame, possibly within strata.
- %>%: the "pipe" operator is used to connect multiple verb actions into a pipeline.

dplyr package also includes a number of data types.

#### Common dplyr function properties

1. The first argument is a data frame
2. The subsequent arguments describe what to do with the data frame, and you can refer to columns in the data frame directly without the $ operator.
3. The return result of the function is a new data frame.
4. Data frames must be properly formatted and annotated to be useful. Data should be tidy: one observation per row, each column with a feature or characteristic of observation.

## Installing dplyr package

Can be installed from CRAN or GitHub uisng devtools pacakage and install_github() fuction. GitHub usually has latest updates of function.
To install from CRAN, run

```
install.packages("dplyr")
```

To install from GitHub use

```
install_github("hadley/dplyr")
```

Remember to load into R session with library()!

```
library(dplyr)
```

If R gives warnings about functions with the same name, ignore for now.

## select()

Examples in this section will use dataset available from textbook's website. Download an unzip then load using readRDS()

```
chicago <- readRDS("chicago.rds")
# Basic characteristics
dim(chicago)
str(chicago)
```

select() function to select columns to focus on. You'll often have large dataset but any given analysis will use subset. Suppose we want first 3 columns. Can use numeric index or column names directly.

```
names(chicago)[1:3]
subset <- select(chicago,city:dptp)
head(subset)
# Note that the : normally cannot be used with names of strings, but inside select() can specify range
# Can also omit variables using negative sign.
select(chicago, -(city:dptp))
# Indicates every variable except city through dptp. Equivalent code:
i <- match("city",names(chicago))
j <- match("dptp", names(chicago))
head(chicago[,-(i:j)]
```

Select also lets special syntax for patterns

```
# keep var ending with "2"
subset <- select(chicago,ends_with("2"))
str(subset)
# or keeping every variable that starts with "d"
subset <- select(chicago, starts_with("d))
str(subset)
```

You can also use general regex.

## filter()

Extract subset rows. Similar to subset() but faster. Extract rows with PM2.5 greater than 30

```
chic.f <- filter(chicago,pm25tmean2 > 30)
str(chic.f)
summary(chic.f$pm25tmean2)

# Can extract arbitrarily complex logical sequence inside of filter()

chic.f <- filter(chicago, pm25tmean > 30 & tmpd > 80)
select(chic.f, date, tmpd, pm25tmean2)
```

## arrange()

Used to reorder rows according to one of the variables/columns. Reordering, while preserving order of other columns, is a pain in R. arrange() simplifies the process.

```
chicago <- arrange(chicago, date) #starts at earliest, ends at oldest
head(select(chicago, date, pm25tmean2), 3)
tail(select(chicago, date, pm25tmean2), 3)

# Columns can be arranged in descending order too using desc() operator
chicago <- arrange(chicago, desc(date))
head(select(chicago, date, pm25tmean2),3)
tail(select(chicago, date, pm25tmean2),3)
```

## rename()

Renaming a variable in R is surprisingly hard to do! rename() makes it easier.

```
head(chicago[, 1:5],3) # names of first 5 variables
# dptp represents dew point temp
# pm25tmean2 provides PM2.5 data
# The given names are obscure and can be renamed
chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
head(chicago[,1:5],3)
```

## mutate()

Compute transformations of variables, such as deriving new variables from existing. For example, can 'detrend' data by subtracting a mean to see how it differs from average rather than absolute level.

```
chicago <- mutate(chicaago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
head(chicago)
```

transmute() does the same as mutate but then *drops all non-transformed variables*

```
head(transmute(chicago,
  pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
  o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
```

## group_by()

Creates summary statistics from the data frame within strata defined by a variable. For example, can compute average annual level of PM2.5. Stratum is the year, derived from date. Standard operation is split data frame into separate pieces defined by variable or group of variables with group_by(), then apply summary function.

```
chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900) # year variable
years <- group_by(chicago, year) # create separate data frame that splits original DF by year.
# Finally, compute summary statistics
summarize(years), pm25 = mean(pm25, na.rm = TRUE),
  o3 = max(o3tmean2,na.rm = TRUE),
  no2 = median(no2tmean2,na.rm = TRUE),
  .groups = "drop")
```

summarize() returns a dataframe with year as first column, then annual averages of pm25, o2, and no2. As a more complicated example, might want to know average levels of ozone and nitrogen dioxide within quintiles of pm25. A regression model would be slicker, but we can do quickly with group_by() and summarize().

```
qq <- quantile(chicago$pm25,seq(0,1,0.2),na.rm = TRUE)
chicago <- utate(chicago, pm25.quint = cut(pm25,qq))
# Now we can group the data frame by the pm.25 quint variable.
quint <- group_by(chicago, pm25.quint)
# Finally, compute mean within quintiles.
summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),
  no2 = mean(no2tmean2, na.rm = TRUE),
  .groups = "drop")
```

From the table above, there isn't a strong relationship between pm25 and o3, but a positive correlation between pm25 and no2. More sophisticated stat modeling can give precise answers to these questions, but this application can get you most of the way there.

### %>%

The pipeline operator is handy for stringing together multiple functions in a sequence of operations. Above, we had to nest the functions when applying multiple. The pipeline operator lets you string operations left to right instead of nesting. In the last section we had to:

1. create a new variable pm25.quint
2. split the data frame by that new variable
3. compute the mean of o3 and no2 in the sub-groups defined by pm25.quint

That can be done with the following sequene:

```
mutate(chicago, pm25.quint = cut(pm25,qq)) %>%
  group_by(pm25.quint) %>%
  summarize(o3 = mean(o3tmean2, na.rm = TRUE),
    no2 = mean(no2tmean2, na.rm = TRUE),
    .groups = "drop")
```

This helps us not create a set of temporary variables along the way or massive nested sequences. The first argument of a pipeline is taken to be the output of the previous element in the pipeline. Another example is average pollutant level by month.

```
mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
  group_by(month) %>%
  summarize(pm25 = mean(pm25, na.rm = TRUE),
    o3 = max(o3tmean2, na.rm = TRUE),
    no2 = median(no2tmean2, na.rm = TRUE),
    .groups = "drop)
```

### Summary

the dplyr package provides a concise set of operations for managing data frames. Additonal benefits:

- can work with other data frame "backends" such as SQL databases. There is an SQL interface for relational databases via the DBI package
- can be integrated with the data.table package for large fast tables.

## Control Structures

Control flow of execution, put "logic" into code. Commonly used strutures:

- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop while a condition is true
- repeat: execute an infinite loop (must break to stop)

- break: break execution of loop
- next: skip an interation

Not really used in interactive sessions, but when writing functions and long expressions.

### if-else

Common conditional Does nothing if condition is false, if you want an action when false, use else. Can have series of tests:

### for loops

Takes an iterator variable and assign it successive values from a sequence or vector. Commonly used to iterate over elements of an object.

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

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

This loop takes the variable i in each iteration of the loop and gives it values 1, 2, 3...10 executes the code, then exits. The following three loops have the same effect.

```
x <- c("a", "b", "c", "d")
for(i in 1:4) {
  print(x[i])
}
```

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

```
# seq_along() is commonly used in conjunction with for loops to generate
# integer sequence based on length of object.
for(i in seq_along(x)) {
  print(x[i])
}
```

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

```
# Not necessary to use an index-type variable.
for(letter in x) {
  print(letter)
}
```

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

```
# For one line loops, can skip braces
for(i in 1:4) print(x[i])
```

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

Not a bad idea to always use curley braces - makes it easier to expand function at later date.

### Nested for loops

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))){
    print(x[i,j])
  }
}
```

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

Commonly used for multidimensional or hierarcharal data structures. Avoid more than 2-3 levels of nesting as it can make code difficult to read. Consider breaking up loops with functions.

### while loops

Begin by testing a condition, if true body executes.

```
count <- 0
while(count < 10) {
  print(count)
  count <- count + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

Can create infinite loops if not careful! Can have multiple conditions, evaluated left to right.

```
z <- 5
set.seed(1)
while(z >= 3 && z <=10) {
  coin <- rbinom(1,1,0.5)
  if(coin == 1) { # random walk
      z <- z + 1
    } else {
      z <- z - 1
    }
}
print(z)
```

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

### repeat loops

Creates an infinite loop, not commonly used in stats applications. Can only be exited with a break Possible paradigm might be iterative algorithm where you are searching for a solution and don't want to stop till you're close enough. Often don't know how many iterations itll take to get close enough.

```
x0 <- 1
tol <- 1e-8

repeat {
  x1 <- computeEstimate()

  if(abs(x1 - x0) < tol) { #close enough?
    break
  } else {
    x0 <- x1
  }
}
```

Above code won't run if the computeEstimate() function is not defined. Looping this way is dangerous as no guarantee itll stop. Can set hard limit on number of iterations using a for loop and then report on convergence.

### next, break

next skips an iteration

```
for(i in 1:100) {
  if(i <=20) {
    next
  }
  # do something else
}
# break exits immediately
for(i in 1:100) {
  print(i)
  if(i > 20) {
    break
  }
}
```

## Functions

Core activity of R programmer. Step from user to developing new functionality for R. Encapsulate sequence of expressions that need to be excuted numerous times, perhaps under different conditions. Often used when code is shared with others or public.
Allows develper to create interfact to code, explicitly specified with set of parameters. Provides abstraction of code to user to simplify process so user doesn't have to know every detail of code. Can allow developer to communicate to user aspects of code that are important or relevant.

### Functions in R

'first class objects' treated like any other R object.

- Functions can be passed as arguments to other functions.
- Can be nested.

### Your first function

Defined using function() directive and stored as R objects of class function.

```
f <- function() {
  # This is an empty function
}
class(f)
```

```
## [1] "function"
```

```
f()
```

```
## NULL
```

Nontrivial example:

```
f <- function() {
  cat("Hello, world! \n")
}
f()
```

```
## Hello, world!
```

Can include arguments that user may explicitly set.

```
f <- function(num) {
  for(i in seq_len(num)) {
    cat("Hello, world!\n")
  }
}
f(3)
```

```
## Hello, world!
## Hello, world!
## Hello, world!
```

Can also return objects

```
f <- function(num) {
  hello <- "Hello, world!\n"
  for(i in seq_len(num)) {
    cat(hello)
  }
  chars <- nchar(hello) * num
  chars
}
meaningoflife <- f(3)
```

```
## Hello, world!
## Hello, world!
## Hello, world!
```

```
print(meaningoflife)
```

```
## [1] 42
```

Return value of a function is always the very last expression that is evaluated. There is a return() function to explictly return values, rarely used in R. If num wasn't specified above, R will give an error. Can set default. formals() will return list of all the formal arguments of a function. Functions have named arguments that can have default values so you can specify argument by name. Useful if has lots of arguments.

## Argument matching

Arguments of functions can be matched by position or by name.

```
str(rnorm)
```

```
## function (n, mean = 0, sd = 1)
```

```
mydata <- rnorm(100, 2, 1)
```

When specifying arguments by name, order doesn't matter. You can mix positional and name matching. Once an argument is matched by name, it is 'taken out' of the argument list and the remaining unnamed arguments are matched in the listed order.

```
args(lm)
# Following two calls are equivalent
lm(data = mydata, y~x, model = FALSE, 1:100)
lm(y~x, mydata, 1:100, model = FALSE)
```

Arguments can also be partially matched.

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Partial matching should be avoided when writing long code or programs as it can lead to confusion. Useful for interactive use.

## Lazy Evaluation

Arguments to functions are evaluated lazily, so only evaluated as needed in body of function. For example:

```
f <- function(a,b) {
  a^2
}
f(2)
```

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

This function never uses argument b, so calling f(2) will not produce an error because 2 is positionally matched to a.

## The ... argument

Indicates a variable number of arguments that are usually passed onto other functions. Often used when extending another function and you don't want to copy the entire argument list of the original function. Example: custom plotting

```r
myplot <- function(x,y,type='l',...) {
  plot(x,y,type = type,...) # passes '...' to 'plot' function
}
```

Generic functions use ... so that extra arguments can be passed to methods.

```r
  mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x000002880bd1c238>
## <environment: namespace:base>
```

The ... argument is necessary when the number of arguments passed to the function cannot be known in advance. Clear functions like paste() and cat().