Unit 2: Data input/output and webscraping

August 28, 2019

References:

- Adler
- Nolan and Temple Lang, XML and Web Technologies for Data Sciences with R.
- Chambers
- R intro manual on CRAN (R-intro).
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies.
- R Data Import/Export manual on CRAN (R-data).
- SCF tutorial on "Working with large datasets in SQL, R, and Python", available from http://statistics.berkeley.edu/computing/training/tutorials.

1 Data storage and formats (outside R)

At this point, we're going to turn to getting data, reading data in, writing data out to disk, and webscraping. We'll focus on doing these manipulations in R, but the concepts and tools involved are common to other languages, so familarity with these in R should allow you to pick up other tools more easily. The main downside to working with datasets in R (true for Python as well) is that the entire dataset resides in memory, so R is not so good for dealing with very large datasets. More on alternatives in a bit. Another common frustration is controlling how the variables are interpreted (numeric, character, factor) when reading data into a data frame.

R has the capability to read in a wide variety of file formats. Let's get a feel for some of the common ones.

1. Flat text files (ASCII files): data are often provided as simple text files. Often one has one record or observation per row and each column or field is a different variable or type of information about the record. Such files can either have a fixed number of characters in each field (fixed width format) or a special character (a delimiter) that separates the fields in each row. Common delimiters are tabs, commas, one or more spaces, and the pipe (|). Common file extensions are *.txt* and *.csv*. Metadata (information about the data) are often stored in a separate file. CSV files are quite common, but if you have files where the data contain commas, other delimiters can be good. Text can be put in quotes in CSV files, and this can allow use of commas within the data. This is difficult to deal with in bash, but *read.table()* in R handles this situation.

    - ASCII is a text format that has $2^7 = 128$ characters and control codes; basically what you see on a standard US keyboard.
    - More on ASCII and extended formats (e.g., UTF-8) that can handle a wider variety of characters in Section 5.
    - One occasionally tricky difficulty is as follows. If you have a text file created in Windows, the line endings are coded differently than in UNIX (a newline (the ASCII character \n) and a carriage return (the ASCII character \r) in Windows vs. only a newline in UNIX). There are UNIX utilities (*fromdos* in Ubuntu, including the SCF Linux machines and *dos2unix* in other Linux distributions) that can do the necessary conversion. If you see *^M* at the end of the lines in a file, that's the tool you need. Alternatively, if you open a UNIX file in Windows, it may treat all the lines as a single line. You can fix this with *todos* or *unix2dos*.

2. In some contexts, such as textual data and bioinformatics data, the data may in a text file with one piece of information per row, but without meaningful columns/fields.

3. In scientific contexts, netCDF (*.nc*) (and the related HDF5) are popular format for gridded data that allows for highly-efficient storage and contains the metadata within the file. The basic structure of a netCDF file is that each variable is an array with multiple dimensions (e.g., latitude, longitude, and time), and one can also extract the values of and metadata about each dimension. The *ncdf4* package in R nicely handles working with netCDF files. These are examples of a binary format, which is not (easily) human readable but can be more space-efficient and faster to work with (because it can allow random access into the data rather than requiring sequential reading). Binary files store information as sequence of bytes, with the meaning of the bytes depending on the specific binary format being used. (Recall that a bit is a single value in base 2 (i.e., a 0 or a 1), while a byte is 8 bits.)

4. Data may also be in text files in formats designed for data interchange between various languages, in particular XML or JSON. These formats are "self-describing"; namely the metadata is part of the file. The *XML2, rvest*, and *jsonlite* packages are useful for reading and writing from these formats.

5. You may be scraping information on the web, so dealing with text files in various formats, including HTML. The *XML2* and *rvest* packages are also useful for reading HTML.

6. Data may already be in a database or in the data storage of another statistical package (*Stata*, *SAS*, *SPSS*, etc.). The *foreign* package in R has excellent capabilities for importing Stata (*read.dta()*), SPSS (*read.spss()*), and SAS (*read.ssd()* and, for XPORT files, *read.xport()*), among others.

7. For Excel, there are capabilities to read an Excel file (see the *readxl* and *XLConnect* package among others), but you can also just go into Excel and export as a CSV file or the like and then read that into R. In general, it's best not to pass around data files as Excel or other spreadsheet format files because (1) Excel is proprietary, so someone may not have Excel and the format is subject to change, (2) Excel imposes limits on the number of rows, (3) one can easily manipulate text files such as CSV using UNIX tools, but this is not possible with an Excel file, (4) Excel files often have more than one sheet, graphs, macros, etc., so they're not a data storage format per se.

8. R can easily interact with databases (SQLite, PostgreSQL, MySQL, Oracle, etc.), querying the database using SQL and returning results to R. More in the big data unit and in the large datasets tutorial mentioned above.

2 Reading data from text files into R

2.1 Core R functions

read.table() is probably the most commonly-used function for reading in data. It reads in delimited files (*read.csv()* and *read.delim()* are special cases of *read.table()*). The key arguments are the delimiter (the *sep* argument) and whether the file contains a header, a line with the variable names. We can use *read.fwf()* to read from a fixed width text file into a data frame.

The most difficult part of reading in such files can be dealing with how R determines the classes of the fields that are read in. There are a number of arguments to *read.table()* and *read.fwf()* that allow the user to control the classes. One difficulty is that character and numeric fields are

sometimes read in as factors. Basically *read.table()* tries to read fields in as numeric and if it finds non-numeric and non-NA values, it reads in as a factor. This can be annoying.

Let's work through a couple examples. Before we do that, let's look at the arguments to *read.table()*. Note that *sep=" "* separates on any amount of white space. In the code chunk below, I've told *knitr* not to print the output to the PDF; you can see the full output by running the code yourself.

```
dat <- read.table(file.path('..', 'data', 'RTADataSub.csv'),
                  sep = ',', head = TRUE)
sapply(dat, class)
## whoops, there is an 'x', presumably indicating missingness:
levels(dat[ ,2])
## let's treat 'x' as a missing value indicator
dat2 <- read.table(file.path('..', 'data', 'RTADataSub.csv'),
                   sep = ',', head = TRUE,
   na.strings = c("NA", "x"), stringsAsFactors = FALSE)
unique(dat2[ ,2])
## hmmm, what happened to the blank values this time?
which(dat[ ,2] == "")
dat2[which(dat[, 2] == "")[1], ] # pull out a line with a missing string

# using 'colClasses'
sequ <- read.table(file.path('..', 'data', 'hivSequ.csv'),
  sep = ',', header = TRUE,
  colClasses = c('integer','integer','character',
    'character','numeric','integer'))
## let's make sure the coercion worked - sometimes R is obstinant
sapply(sequ, class)
## that made use of the fact that a data frame is a list
```

Note that you can avoid reading in one or more columns by specifying *NULL* as the column class for those columns to be omitted. Also, specifying the *colClasses* argument explicitly should make for faster file reading. Finally, setting `stringsAsFactors=FALSE` is standard practice. You can set that by default to apply generally in your *.Rprofile* using `options(stringsAsFactors = FALSE)`. Or use *readr::read_csv()* as discussed below.

If possible, it's a good idea to look through the input file in the shell or in an editor before reading into R to catch such issues in advance. Using *less* on *RTADataSub.csv* would have revealed

these various issues, but note that *RTADataSub.csv* is a 1000-line subset of a much larger file of data available from the kaggle.com website. So more sophisticated use of UNIX utilities as we saw in Unit 2 is often useful before trying to read something into R.

The basic function *scan()* simply reads everything in, ignoring lines, which works well and very quickly if you are reading in a numeric vector or matrix. *scan()* is also useful if your file is free format - i.e., if it's not one line per observation, but just all the data one value after another; in this case you can use *scan()* to read it in and then format the resulting character or numeric vector as a matrix with as many columns as fields in the dataset. Remember that the default is to fill the matrix by column.

If the file is not nicely arranged by field (e.g., if it has ragged lines), we'll need to do some more work. *readLines()* will read in each line into a separate character vector, after which we can process the lines using text manipulation. Here's an example from some US meteorological data where I know from metadata (not provided here) that the 4-11th values are an identifier, the 17-20th are the year, the 22-23rd the month, etc.

```
dat <- readLines(file.path('..', 'data', 'precip.txt'))
id <- as.factor(substring(dat, 4, 11) )
year <- substring(dat, 18, 21)
year[1:5]

## [1] "2010" "2010" "2010" "2010" "2010"

class(year)

## [1] "character"

year <- as.integer(substring(dat, 18, 21))
month <- as.integer(substring(dat, 22, 23))
nvalues <- as.integer(substring(dat, 28, 30))
```

Note that for *precip.txt*, reading in using *read.fwf()* would be a good strategy.

R allows you to read in not just from a file but from a more general construct called a *connection*. Here are some examples of connections:

```
dat <- readLines(pipe("ls -al"))
dat <- read.table(pipe("unzip dat.zip"))
dat <- read.csv(gzfile("dat.csv.gz"))
dat <- readLines("http://www.stat.berkeley.edu/~paciorek/index.html")
```

In some cases, you might need to create the connection using *url()* or using the *curl()* function from the *curl* package. Though for the example here, simply passing the URL to *readLines()* does work. (In general, *curl::curl()* provides some nice features for reading off the internet.)

```
wikip1 <- readLines("https://wikipedia.org")
wikip2 <- readLines(url("https://wikipedia.org"))
library(curl)
wikip3 <- readLines(curl("https://wikipedia.org"))
```

If a file is large, we may want to read it in in chunks (of lines), do some computations to reduce the size of things, and iterate. *read.table()*, *read.fwf()* and *readLines()* all have the arguments that let you read in a fixed number of lines. To read-on-the-fly in blocks, we need to first establish the connection and then read from it sequentially.

```
con <- file(file.path("..", "data", "precip.txt"), "r")
## "r" for 'read' - you can also open files for writing with "w"
## (or "a" for appending)
class(con)
blockSize <- 1000 # obviously this would be large in any real application
nLines <- 300000
for(i in 1:ceiling(nLines / blockSize)){
    lines <- readLines(con, n = blockSize)
    # manipulate the lines and store the key stuff
}
close(con)
```

Here's an example of using *curl()* to do this for a file on the web.

```
URL <- "https://www.stat.berkeley.edu/share/paciorek/2008.csv.gz"
con <- gzcon(curl(URL, open = "r"))
## url() in place of curl() works too
for(i in 1:8) {
        print(i)
        print(system.time(tmp <- readLines(con, n = 100000)))
        print(tmp[1])
}

## [1] 1
```

```
##    user  system elapsed
##   0.750   0.002   0.752
## [1] "Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTi
## [1] 2
##    user  system elapsed
##   0.827   0.013   0.840
## [1] "2008,1,29,2,1938,1935,2308,2257,XE,7676,N11176,150,142,104,11,3,SLC,
## [1] 3
##    user  system elapsed
##   0.804   0.000   0.805
## [1] "2008,1,20,7,1540,1525,1651,1637,OO,5703,N227SW,71,72,58,14,15,SBA,S
## [1] 4
##    user  system elapsed
##   0.814   0.000   0.815
## [1] "2008,1,2,3,1313,1250,1443,1425,WN,440,N461WN,150,155,138,18,23,MCO,S
## [1] 5
##    user  system elapsed
##   0.760   0.001   0.759
## [1] "2008,1,24,4,1026,1015,1116,1110,MQ,3926,N653AE,50,55,38,6,11,MLI,ORD
## [1] 6
##    user  system elapsed
##   0.747   0.000   0.747
## [1] "2008,1,4,5,1129,1125,1352,1350,AA,1145,N438AA,203,205,187,2,4,ORD,SI
## [1] 7
##    user  system elapsed
##   0.751   0.008   0.759
## [1] "2008,1,10,4,716,720,1025,1024,DL,1590,N991DL,129,124,107,1,-4,AUS,AT
## [1] 8
##    user  system elapsed
##   0.771   0.000   0.771
## [1] "2008,2,15,5,2127,2132,2254,2312,XE,7663,N33182,87,100,71,-18,-5,SLC,

close(con)
```

More details on sequential (on-line) processing of large files can be found in the tutorial on large datasets mentioned in the reference list above.

One cool trick that can come in handy is to create a *text connection*. This lets you 'read' from

an R character vector as if it were a text file and could be handy for processing text. For example, you could then use *read.fwf()* applied to *con*.

```
dat <- readLines('../data/precip.txt')
con <- textConnection(dat[1], "r")
read.fwf(con, c(3,8,4,2,4,2))

##    V1      V2   V3 V4   V5 V6
## 1 DLY 1000807 PRCP HI 2010  2
```

We can create connections for writing output too. Just make sure to open the connection first.

2.2 File paths

A few notes on file paths, related to ideas of reproducibility.

1. In general, you don't want to hard-code absolute paths into your code files because those absolute paths won't be available on the machines of anyone you share the code with. Instead, use paths relative to the directory the code file is in, or relative to a baseline directory for the project, e.g.:

```
dat <- read.csv('../data/cpds.csv')
```

2. Be careful with the directory separator in Windows files: you can either do *"C:\\mydir\\file.txt"* or *"C:/mydir/file.txt"*, but not *"C:\mydir\file.txt"*, and note the next comment about avoiding use of '\\' for portability.

3. Using UNIX style directory separators will work in Windows, Mac or Linux, but using Windows style separators is not portable across operating systems.

```
## good: will work on Windows
dat <- read.csv('../data/cpds.csv')
## bad: won't work on Mac or Linux
dat <- read.csv('..\\data\\cpds.csv')
```

4. Even better, use *file.path()* so that paths are constructed specifically for the operating system the user is using:

```
## good: operating-system independent
dat <- read.csv(file.path('..', 'data', 'cpds.csv'))
```

2.3 The *readr* package

readr is intended to deal with some of the shortcomings of the base R functions, such as defaulting to `stringsAsFactors=FALSE`, leaving column names unmodified, and recognizing dates/times. It reads data in much more quickly than the base R equivalents. See this blog post. Some of the readr functions that are analogs to the comparably-named base R functions are *read_csv()*, *read_fwf()*, *read_lines()*, and *read_table()*.

Let's try out *read_csv()* on the airline dataset used in the R bootcamp.

```
library(readr)

##
## Attaching package:  'readr'
## The following object is masked from 'package:curl':
##
##     parse_date

## I'm violating the rule about absolute paths here!!
## (airline.csv is big enough that I don't want to put it in the
##     course repository)
setwd('~/staff/workshops/r-bootcamp-2018/data')
system.time(dat <- read.csv('airline.csv', stringsAsFactors = FALSE))

##    user  system elapsed
##   3.756   0.203   3.967

system.time(dat2 <- read_csv('airline.csv'))

## Parsed with column specification:
## cols(
##   .default = col_double(),
```

```
##   UniqueCarrier = col_character(),
##   TailNum = col_character(),
##   Origin = col_character(),
##   Dest = col_character(),
##   CancellationCode = col_character()
## )
## See spec(...)  for full column specifications.
```

2.4 Reading data quickly

In addition to the tips above, there are a number of packages that allow one to read large data files quickly, in particular *data.table*, *ff*, and *bigmemory*. In general, these provide the ability to load datasets into R without having them in memory, but rather stored in clever ways on disk that allow for fast access. Metadata is stored in R. More on this in the unit on big data and in the tutorial on large datasets mentioned in the reference list above.

3 Webscraping and working with XML and JSON

The new (well, as of 2015) book *XML and Web Technologies for Data Sciences with R* by Deb Nolan (UCB Stats faculty) and Duncan Temple Lang (UCB Stats PhD alumnus and UC Davis Stats faculty) provides extensive information about getting and processing data off of the web, including interacting with web services such as REST and SOAP and programmatically handling authentication.

Here are some UNIX command-line tools to help in webscraping and working with files in formats such as JSON, XML, and HTML: http://jeroenjanssens.com/2013/09/19/seven-command-line-tools-for-data-science.html.

We'll cover a few basic examples in this section, but HTML and XML formatting and navigating the structure of such pages is beyond the scope of what we can cover in detail. The key thing is to know that the tools exist so that you can learn how to use them if faced with such formats.

3.1 Reading HTML

Let's see a brief example of reading in HTML tables. One lesson here is not to write a lot of your own code to do something that someone else has probably already written a package for. We'll use the *rvest* package (you can also see usage of the now-unmaintained *xml* package in the code file for this unit).

```
library(rvest)  # uses xml2
```

```
## Loading required package:  xml2
##
## Attaching package:  'rvest'
## The following object is masked from 'package:readr':
##
##    guess_encoding
```

```
URL <- "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_
html <- read_html(URL)
tbls <- html_table(html_nodes(html, "table"))
sapply(tbls, nrow)
```

```
## [1] 240  12
```

```
pop <- tbls[[2]]
head(pop)
```

```
##   vteLists of countries by population statistics
## 1                                         Global
## 2                              (Sub-)continents
## 3                              Intercontinental
## 4                          Cities/urban regions
## 5                               Past and future
## 6                            Population density
##
## 1
## 2
## 3                                                Americas\nArab world\nComm
## 4                                                                National ca
## 5 Past and future population\nWorld population estimates\n1\n1000\n1500\n
## 6                                                  Current density\nPast a
```

read_html() works by reading in the HTML as text and then parsing it to build up a tree containing the HTML elements. Then *html_nodes()* finds the HTML tables and *html_table()* converts them to data frames. rvest is part of the tidyverse, so it's often used with piping, e.g.,

```
library(magrittr)
tbls <- URL %>% read_html("table") %>% html_table()
```

It's often useful to be able to extract the hyperlinks in an HTML document. We'll find the link using CSS selectors, which allow you to search for elements within HTML:

```
URL <- "http://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year"
## approach 1: search for elements with href attribute
links <- read_html(URL) %>% html_nodes("[href]") %>% html_attr('href')
## approach 2: search for HTML 'a' tags
links <- read_html(URL) %>% html_nodes("a") %>% html_attr('href')
head(links, n = 10)

##  [1] "?C=N;O=D"              "?C=M;O=A"
##  [3] "?C=S;O=A"              "?C=D;O=A"
##  [5] "/pub/data/ghcn/daily/" "1763.csv.gz"
##  [7] "1764.csv.gz"           "1765.csv.gz"
##  [9] "1766.csv.gz"           "1767.csv.gz"
```

More generally, we may want to read an HTML document, parse it into its components (i.e., the HTML elements), and navigate through the tree structure of the HTML. Here we use the *XPath* language to specify elements rather than CSS selectors. XPath can also be used for navigating through XML documents.

```
## find all 'a' nodes that have attribute href; then
## extract the 'href' attribute
links <- read_html(URL) %>% html_nodes(xpath = "//a[@href]") %>%
    html_attr('href')
head(links)

## [1] "?C=N;O=D"              "?C=M;O=A"
## [3] "?C=S;O=A"              "?C=D;O=A"
## [5] "/pub/data/ghcn/daily/" "1763.csv.gz"
```

```
## we can extract various information
listOfANodes <- read_html(URL) %>% html_nodes(xpath = "//a[@href]")
listOfANodes %>% html_attr('href') %>% head(n = 10)

##  [1] "?C=N;O=D"              "?C=M;O=A"
##  [3] "?C=S;O=A"              "?C=D;O=A"
##  [5] "/pub/data/ghcn/daily/" "1763.csv.gz"
##  [7] "1764.csv.gz"           "1765.csv.gz"
##  [9] "1766.csv.gz"           "1767.csv.gz"

listOfANodes %>% html_name() %>% head(n = 10)

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

listOfANodes %>% html_text()  %>% head(n = 10)

##  [1] "Name"             "Last modified"
##  [3] "Size"             "Description"
##  [5] "Parent Directory" "1763.csv.gz"
##  [7] "1764.csv.gz"      "1765.csv.gz"
##  [9] "1766.csv.gz"      "1767.csv.gz"
```

Here's another example of extracting specific components of information from a webpage.

```
URL <- "https://www.nytimes.com"
headlines <- read_html(URL) %>% html_nodes("h2") %>% html_text()
head(headlines)
```

We can explore the underlying HTML source in advance of writing our code by looking at the page source directly in the browser (e.g., in Firefox see `Web Developer -> Page Source` and in Chrome `More tools -> Developer tools`), or by downloading the webpage and looking at it in an editor, although in some cases (such as the nytimes.com case), what we might see is a lot of JavaScript.

3.2 XML

XML is a markup language used to store data in self-describing (no metadata needed) format, often with a hierarchical structure. It consists of sets of elements (also known as nodes because

they generally occur in a hierarchical structure and therefore have parents, children, etc.) with tags that identify/name the elements, with some similarity to HTML. Some examples of the use of XML include serving as the underlying format for Microsoft Office and Google Docs documents and for the KML language used for spatial information in Google Earth.

Here's a brief example. The book with id attribute *bk101* is an element; the author of the book is also an element that is a child element of the book. The id attribute allows us to uniquely identify the element.

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications with XML.</des
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
     <description>A former architect battles corporate zombies, an evil sor
   </book>
</catalog>
```

We can read XML documents into R using `xml2::read_xml()` and then manipulate it using other functions from the *xml2* package. Here's an example of working with lending data from the Kiva lending non-profit. You can see the XML format in a browser at

http://api.kivaws.org/v1/loans/newest.xml.

XML documents have a tree structure with information at nodes. As above with HTML, one can use the *XPath* language for navigating the tree and finding and extracting information from the node(s) of interest. Here is some example code for extracting loan info from the Kiva data.

```r
library(xml2)
doc <- read_xml("https://api.kivaws.org/v1/loans/newest.xml")
data <- as_list(doc)
names(data)
```

```
## [1] "response"
```

```r
names(data$response)
```

```
## [1] "paging" "loans"
```

```r
length(data$response$loans)
```

```
## [1] 20
```

```r
data$response$loans[[2]][c('name', 'activity',
                           'sector', 'location', 'loan_amount')]
```

```
## $name
## $name[[1]]
## [1] "Naumi"
##
##
## $activity
## $activity[[1]]
## [1] "Poultry"
##
##
## $sector
## $sector[[1]]
## [1] "Agriculture"
##
##
## $location
## $location$country_code
## $location$country_code[[1]]
## [1] "KE"
##
```

```
##
## $location$country
## $location$country[[1]]
## [1] "Kenya"
##
##
## $location$town
## $location$town[[1]]
## [1] "Nandi Hills"
##
##
## $location$geo
## $location$geo$level
## $location$geo$level[[1]]
## [1] "town"
##
##
## $location$geo$pairs
## $location$geo$pairs[[1]]
## [1] "0.103073 35.176372"
##
##
## $location$geo$type
## $location$geo$type[[1]]
## [1] "point"
##
##
##
##
## $loan_amount
## $loan_amount[[1]]
## [1] "300"

## alternatively, extract only the 'loans' info (and use pipes)
loansNode <- doc %>% xml_nodes('loans')
loanInfo <- loansNode %>% xml_children() %>% as_list()
```

```r
length(loanInfo)
```

```
## [1] 20
```

```r
names(loanInfo[[1]])
```

```
##  [1] "id"
##  [2] "name"
##  [3] "description"
##  [4] "status"
##  [5] "funded_amount"
##  [6] "basket_amount"
##  [7] "image"
##  [8] "activity"
##  [9] "sector"
## [10] "use"
## [11] "location"
## [12] "partner_id"
## [13] "posted_date"
## [14] "planned_expiration_date"
## [15] "loan_amount"
## [16] "borrower_count"
## [17] "lender_count"
## [18] "bonus_credit_eligibility"
## [19] "tags"
```

```r
names(loanInfo[[1]]$location)
```

```
## [1] "country_code" "country"      "town"
## [4] "geo"
```

```r
## suppose we only want the country locations of the loans (using XPath)
xml_find_all(loansNode, '//location//country') %>% xml_text()
```

```
##  [1] "Paraguay"     "Kenya"        "Paraguay"
##  [4] "Lebanon"      "Egypt"        "Indonesia"
##  [7] "Burkina Faso" "Philippines"  "Philippines"
## [10] "Philippines"  "Palestine"    "Uganda"
```

```
## [13] "Uganda"       "Palestine"    "Kenya"
## [16] "Mozambique"   "Uganda"       "Kenya"
## [19] "Mozambique"   "Georgia"

## or extract the geographic coordinates
xml_find_all(loansNode, '//location//geo/pairs')

## {xml_nodeset (20)}
##  [1] <pairs>-24.194867 -56.561647</pairs>
##  [2] <pairs>0.103073 35.176372</pairs>
##  [3] <pairs>-24.665338 -56.455192</pairs>
##  [4] <pairs>33.870763 35.513959</pairs>
##  [5] <pairs>27 30</pairs>
##  [6] <pairs>-6.178056 106.63</pairs>
##  [7] <pairs>13 -2</pairs>
##  [8] <pairs>10.766667 122.066667</pairs>
##  [9] <pairs>10.306908 119.256946</pairs>
## [10] <pairs>10.766667 122.066667</pairs>
## [11] <pairs>32.464635 35.293859</pairs>
## [12] <pairs>-0.116667 30.483333</pairs>
## [13] <pairs>-0.264241 30.108403</pairs>
## [14] <pairs>32.319405 35.023986</pairs>
## [15] <pairs>0.283333 34.75</pairs>
## [16] <pairs>-26.041667 32.325278</pairs>
## [17] <pairs>-0.116667 30.483333</pairs>
## [18] <pairs>-0.716667 36.433333</pairs>
## [19] <pairs>-26.041667 32.325278</pairs>
## [20] <pairs>42.328499 42.600368</pairs>
```

3.3 Reading JSON

JSON files are structured as "attribute-value" pairs (aka "key-value" pairs), often with a hierarchical structure. Here's a brief example:

```
{
  "firstName": "John",
```

```
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

A set of key-value pairs is a named array and is placed inside braces (squiggly brackets). Note the nestedness of arrays within arrays (e.g., address within the overarching person array and the use of square brackets for unnamed arrays (i.e., vectors of information), as well as the use of different types: character strings, numbers, null, and (not shown) boolean/logical values. JSON and XML can be used in similar ways, but JSON is less verbose than XML.

We can read JSON into R using *fromJSON()* in the *jsonlite* package. Let's play again with the Kiva data. The same data that we had worked with in XML format is also available in JSON format: http://api.kivaws.org/v1/loans/newest.json.

```
library(jsonlite)
data <- fromJSON("http://api.kivaws.org/v1/loans/newest.json")
names(data)

## [1] "paging" "loans"
```

```r
class(data$loans) # nice!
```

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

```r
head(data$loans)
```

```
##        id                          name languages
## 1 1823099 Maria Auxiliadora Poty Group    es, en
## 2 1823579                         Naumi        en
## 3 1823094             Ykua Pindo Group    es, en
## 4 1823580                         Sarah        en
## 5 1823576                        Fatema        en
## 6 1823112                          Iyam        en
##        status funded_amount basket_amount image.id
## 1 fundraising             0            25  3235716
## 2 fundraising             0             0  3236397
## 3 fundraising            25             0  3235704
## 4 fundraising             0             0  3236398
## 5 fundraising             0             0  3236395
## 6 fundraising            50             0  3238115
##   image.template_id                      activity
## 1                 1                         Dairy
## 2                 1                       Poultry
## 3                 1                         Dairy
## 4                 1 Primary/secondary school costs
## 5                 1                   Grocery Store
## 6                 1      Personal Housing Expenses
##        sector
## 1 Agriculture
## 2 Agriculture
## 3 Agriculture
## 4   Education
## 5        Food
## 6     Housing
##
## 1
## 2
```

```
## 3
## 4
## 5 to buy pasta, rice, oils, butter, flour, chips, and chocolates to incre
## 6
##   location.country_code location.country location.town
## 1                    PY         Paraguay     San Pedro
## 2                    KE            Kenya   Nandi Hills
## 3                    PY         Paraguay       SantanÃ
## 4                    LB          Lebanon      Tayouneh
## 5                    EG            Egypt      Baniswef
## 6                    ID        Indonesia     Tangerang
##   location.geo.level    location.geo.pairs
## 1               town -24.194867 -56.561647
## 2               town    0.103073 35.176372
## 3               town -24.665338 -56.455192
## 4               town   33.870763 35.513959
## 5            country                27 30
## 6               town      -6.178056 106.63
##   location.geo.type partner_id          posted_date
## 1             point         58 2019-08-28T14:40:02Z
## 2             point        156 2019-08-28T14:30:09Z
## 3             point         58 2019-08-28T14:30:02Z
## 4             point         77 2019-08-28T14:30:02Z
## 5             point        440 2019-08-28T14:20:12Z
## 6             point        406 2019-08-28T14:20:11Z
##   planned_expiration_date loan_amount borrower_count
## 1    2019-09-27T14:40:02Z        2775             15
## 2    2019-09-27T14:30:09Z         300              1
## 3    2019-09-27T14:30:02Z        3375             19
## 4    2019-09-27T14:30:02Z        1000              1
## 5    2019-09-27T14:20:12Z        1225              1
## 6    2019-09-27T14:20:11Z         625              1
##   lender_count bonus_credit_eligibility tags
## 1            0                     TRUE NULL
## 2            0                     TRUE NULL
## 3            1                     TRUE NULL
```

```
## 4             0                      TRUE NULL
## 5             0                     FALSE NULL
## 6             2                     FALSE NULL
##                              themes
## 1                              NULL
## 2                    Rural Exclusion
## 3                              NULL
## 4                             Youth
## 5 Rural Exclusion, Underfunded Areas
## 6               Water and Sanitation
```

One disadvantage of JSON is that it is not set up to deal with missing values, infinity, etc.

3.4 Webscraping ethics and best practices

Before you set up any automated downloading of materials/data from the web you should make sure that what you are about to do is consistent with the rules provided by the website. Some places to look for information on what the website allows are:

- legal pages such as Terms of Service or Terms and Conditions on the website.
- check the robots.txt file (e.g., https://scholar.google.com/robots.txt) to see what a web crawler is allowed to do, and whether the site requires a particular delay between requests to the sites
- potentially contact the site owner if you plan to scrape a large amount of data

Here are some links with useful information:

- A blog post overview on webscraping and robots.txt
- Blog post on webscraping ethics
- Some information on how to understand a robots.txt file

In many cases you will want to include a time delay between your automated requests to a site, including if you are not actually crawling a site but just want to automate a small number of queries.

3.5 Using web APIs to get data

Here we'll see briefly some examples of making requests over the Web to get data. We'll see simple HTTP requests, as well as use APIs to systematically query a website for information

based on a documented interface. The packages *RCurl* and *httr* are useful for a wide variety of such functionality. Note that much of the functionality I describe below is also possible within bash using either *wget* or *curl*.

We've already seen some basic downloading of html from webpages, which use the HTTP request GET.

3.5.1 HTTP requests

Sometime specific information can be downloaded simply by constructing a static URL. Let's look at some UN data (agricultural crop data). By going to http://data.un.org/Explorer.aspx?d=FAO, and clicking on "Crops", we'll see a bunch of agricultural products with "View data" links. Click on "apricots" as an example and you'll see a "Download" button that allows you to download a CSV of the data. Let's select a range of years and then try to download "by hand". Sometimes we can right-click on the link that will download the data and directly see the URL that is being accessed and then one can deconstruct it so that you can create URLs programmatically to download the data you want.

In this case, we can't see the full URL that is being used. More generally, rather than looking at the URL associated with a link we may need to view the actual HTTP request sent by our browser to the server. We can do this using features of the browser (e.g., in Firefox see `Web Developer -> Network` and in Chrome `More tools -> Developer tools -> Network`). Based on this we can see that an HTTP GET request is being used with a URL such as:
http://data.un.org/Handlers/DownloadHandler.ashx?DataFilter=itemCode:526;year:2003,2004,2005,2006,2007&

The stuff at the end of the URL specifies inputs passed to the server separated by '&', in this case relating to the itemCode, dates, output format, etc. So we could more easily download the data using that URL, which we can fairly easily construct using string processing in bash, R, or Python, such as this:

```
## example URL:
##"http://data.un.org/Handlers/DownloadHandler.ashx?DataFilter=
##itemCode:526;year:2003,2004,2005,2006,2007&DataMartId=FAO&
##Format=csv&c=2,3,4,5,6,7&s=countryName:asc"
itemCode <- 526
baseURL <- "http://data.un.org/Handlers/DownloadHandler.ashx"
yrs <- paste(as.character(2003:2007), collapse = ",")
filter <- paste0("?DataFilter=itemCode:", itemCode, ";year:", yrs)
args1 <- "&DataMartId=FAO&Format=csv&c=2,3,4,5,6,7&"
args2 <- "s=countryName:asc,elementCode:asc,year:desc"
```

```
url <- paste0(baseURL, filter, args1, args2)
## if the website provided a CSV we could just do this:
## apricots <- read.csv(url)
## but it zips the file
temp <- tempfile()  ## give name for a temporary file
download.file(url, temp)
dat <- read.csv(unzip(temp))  ## using a connection (see Section 2)

head(dat)

##   Country.or.Area Element.Code        Element Year
## 1     Afghanistan         5312 Area harvested 2007
## 2     Afghanistan         5312 Area harvested 2006
## 3     Afghanistan         5312 Area harvested 2005
## 4     Afghanistan         5312 Area harvested 2004
## 5     Afghanistan         5312 Area harvested 2003
## 6     Afghanistan         5419          Yield 2007
##    Unit Value Value.Footnotes
## 1    ha  8000               F
## 2    ha  8030
## 3    ha  7223              Im
## 4    ha  5200               F
## 5    ha  7007
## 6 hg/ha 72500              Fc
```

A more sophisticated way to do the download is to pass the request in a structured way with named input parameters. This request is easier to construct programmatically. Here what is returned is a zip file, which is represented in R as a sequence of "raw" bytes. We can use httr's *GET()*, followed by writing to disk and reading back in, as follows (for some reason knitr won't print the output...):

```
library(httr)

##
## Attaching package:  'httr'
## The following object is masked from 'package:curl':
##
```

```
##    handle_reset

output2 <- GET(baseURL, query = list(
               DataFilter = paste0("itemCode:", itemCode, ";year:", yrs),
               DataMartID = "FAO", Format = "csv", c = "2,3,4,5,6,7",
               s = "countryName:asc,elementCode:asc,year:desc"))
temp <- tempfile()  ## give name for a temporary file
writeBin(content(output2, 'raw'), temp)  ## write out as zip file
dat <- read.csv(unzip(temp))
head(dat)
```

In some cases we may need to send a lot of information as part of the URL in a GET request. If it gets to be too long (e.g,, more than 2048 characters) many web servers will reject the request. Instead we may need to use an HTTP POST request. A typical request would have syntax like this (using *RCurl*), supposing that the inputs were named *start-year* and *end-year*.

```
if(url.exists('http://www.wormbase.org/db/searches/advanced/dumper')) {
      x = postForm('http://www.wormbase.org/db/searches/advanced/dumper',
               species="briggsae",
               list="",
               flank3="0",
               flank5="0",
```

```
                feature="Gene Models",
                dump = "Plain TEXT",
                orientation = "Relative to feature",
                relative = "Chromsome",
                DNA ="flanking sequences only",
                .cgifields = paste(c("feature", "orientation", "DNA",
                                     "dump","relative"), collapse=", "))
```

httr and *RCurl* can handle other kinds of HTTP requests such as PUT and DELETE. Finally, some websites use cookies to keep track of users and you may need to download a cookie in the first interaction with the HTTP server and then send that cookie with later interactions. More details are available in the Nolan and Temple Lang book.

3.5.2 APIs: REST- and SOAP-based web services

While webscraping with requests such as just described can work well, it was a bit convoluted. We basically needed to deconstruct the queries a browser makes and then mimic that behavior, in some cases having to parse HTML output to get at data. If the webpage changes even a little bit, our carefully constructed query syntax may fail. An alternative is to use a web service specifically designed to serve data or allow other interactions via an Applications Programming Interface (API). REST and SOAP are popular API standards/styles. Both REST and SOAP use HTTP requests; we'll focus on REST as it is more common and simpler.

When using REST, we access *resources*, which might be a Facebook account or a database of stock quotes. The resource may return information in the form of an HTML file or JSON, CSV or something else. REST generally uses XML or JSON as the format for the request (if not a simple GET request) and what is returned.

Let's see an example of accessing climate model output data from the World Bank. The API is documented here: http://data.worldbank.org/developers/climate-data-api. Following that documentation we can download monthly average precipitation predictions for 2080-2099 for the US (ISO3 code 'USA') based on global climate model simulations. In this case what the World Bank refers to as the REST-based query is simply constructing a straightforward URL.

```
times <- c(2080, 2099)
countryCode <- 'USA'
baseURL <- "http://climatedataapi.worldbank.org/climateweb/rest/v1/country"
##" http://climatedataapi.worldbank.org/climateweb/rest/v1/country"
type <- "mavg"
```

```
var <- "pr"
data <- read.csv(paste(baseURL, type, var, times[1], times[2],
                       paste0(countryCode, '.csv'), sep = '/'))
head(data)
```

```
##             GCM var scenario from_year to_year      Jan
## 1   bccr_bcm2_0  pr       a2      2080    2099 67.03573
## 2   bccr_bcm2_0  pr       b1      2080    2099 62.36411
## 3 cccma_cgcm3_1  pr       a2      2080    2099 73.05678
## 4 cccma_cgcm3_1  pr       b1      2080    2099 68.13736
## 5      cnrm_cm3  pr       a2      2080    2099 72.18374
## 6      cnrm_cm3  pr       b1      2080    2099 69.87911
##        Feb      Mar      Apr      May      Jun      Jul
## 1 60.34472 68.55613 69.73249 70.75057 68.87670 75.50839
## 2 57.25621 65.37839 66.83145 71.42986 66.70454 76.21705
## 3 65.88258 69.07827 70.52308 71.27610 71.40891 71.73378
## 4 56.80470 64.87122 64.26042 65.26155 65.63476 68.85459
## 5 64.47102 76.22999 79.40222 94.16236 93.20071 92.61398
## 6 59.47017 70.12903 80.61524 92.51666 92.54483 96.00988
##        Aug      Sep      Oct      Nov      Dec
## 1 79.16541 76.60718 79.72601 72.12957 71.83717
## 2 76.50563 81.48503 73.44661 67.69188 62.61851
## 3 71.16824 72.37070 72.17842 85.11507 77.58228
## 4 68.02093 66.02112 70.71792 77.58472 77.31110
## 5 87.12436 87.24258 86.20070 77.22606 78.49393
## 6 91.05730 88.87449 82.81388 70.25031 71.83962
```

Often what distinguishes an API from what we've discussed in previous sections is that the API documents what information it expects from the user and returns the result in a standard format (e.g., a particular file format rather than producing a webpage). Here we can see the Kiva API, which allows us to construct queries on the Kiva data that we saw some of earlier.

The Nolan and Temple Lang book provides a number of examples of different ways of authenticating with web services that control access to the service.

Finally, some web services allow us to pass information to the service in addition to just getting data or information. E.g., you can programmatically interact with your Facebook, Dropbox, and Google Drive accounts using REST based on HTTP POST, PUT, and DELETE. Authentication is of course important in these contexts and some times you would first authenticate with your login

and password and receive a "token". This token would then be used in subsequent interactions in the same session.

3.5.3 Packaged access to an API

For popular websites/data sources, a developer may have packaged up the API calls in a user-friendly fashion for use from R, Python or other software. For example there are Python (twitter) and R (twitteR) packages for interfacing with Twitter via its API.

Here's some example code for Python (the Python package seems to be more fully-featured than the R package). This looks up the US senators' Twitter names and then downloads a portion of each of their timelines, i.e., the time series of their tweets. Note that Twitter has limits on how much one can download at once.

```
import json
import twitter

# You will need to set the following variables with your
# personal information.  To do this you will need to create
# a personal account on Twitter (if you don't already have
# one).  Once you've created an account, create a new
# application here:
#    https://dev.twitter.com/apps
#
# You can manage your applications here:
#    https://apps.twitter.com/
#
# Select your application and then under the section labeled
# "Key and Access Tokens", you will find the information needed
# below.  Keep this information private.
CONSUMER_KEY       = ""
CONSUMER_SECRET    = ""
OAUTH_TOKEN        = ""
OAUTH_TOKEN_SECRET = ""

auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)
api = twitter.Twitter(auth=auth)
```

```
# get the list of senators
senators = api.lists.members(owner_screen_name="gov", slug="us-senate", cou
with open("senators-list.json", "w") as f:
    json.dump(senators, f, indent=4, sort_keys=True)

# get all the senators' timelines
names = [d["screen_name"] for d in senators["users"]]
timelines = [api.statuses.user_timeline(screen_name=name, count = 500)
             for name in names]
with open("timelines.json", "w") as f:
    json.dump(timelines, f, indent=4, sort_keys=True)
```

3.6 Accessing dynamic pages

Some websites dynamically change in reaction to the user behavior. In these cases you need a tool that can mimic the behavior of a human interacting with a site. Some options are:

- *selenium* (and the *RSelenium* wrapper for R) is a popular tool for doing this.
- *splash* (and the *splashr* wrapper for R) is another approach.

4 File and string encodings

Text (either in the form of a file with regular language in it or a data file with fields of character strings) will often contain characters that are not part of the limited ASCII set of characters, which has $2^7 = 128$ characters and control codes; basically what you see on a standard US keyboard. Each character takes up one byte (8 bits) of space. We can actually hand-generate an ASCII file using the binary representation of each character in R as an illustration.

```
## 39 in hexadecimal is '9'
## 0a is a newline (at least in Linux/Mac)
## 3a is ':'
x <- as.raw(c('0x39','0x0a','0x3a'))  ## i.e., "9\n:" in ascii
x

## [1] 39 0a 3a
```

```
charToRaw('9\n:')

## [1] 39 0a 3a

writeBin(x, 'tmp.txt')
readLines('tmp.txt')

## Warning in readLines("tmp.txt"):  incomplete final line found on
'tmp.txt'

## [1] "9" ":"

system('ls -l tmp.txt', intern = TRUE)

## [1] "-rw-r--r-- 1 paciorek scfstaff 3 Aug 28 07:57 tmp.txt"

system('cat tmp.txt')
```

For non-ASCII files you may need to deal with the text encoding (the mapping of individual characters (including tabs, returns, etc.) to a set of numeric codes). There are a variety of different encodings for text files, with different ones common on different operating systems. UTF-8 is an encoding for the Unicode characters that includes more than 110,000 characters from 100 different alphabets/scripts. It's widely used on the web. Latin-1 encodes a small subset of Unicode and contains the characters used in many European languages (e.g., letters with accents). Here's an example of using a non-ASCII Unicode character:

```
euro <- '\u20ac' # Euro currency symbol as Unicode 'code point'
Encoding(euro)
euro
writeBin(euro, 'tmp2.txt')
system('ls -l tmp2.txt') ## here the euro takes up four bytes
## so the system knows how to interpret the UTF-8 encoded file
## and represent the Unicode character on the screen:
system('cat tmp2.txt')
```

(I have turned off evaluation of that chunk as something strange is going on when I create the PDF.)

UTF-8 is cleverly designed in terms of the bit-wise representation of characters such that ASCII characters still take up one byte, and most other characters take two bytes, but some take four bytes.

The UNIX utility *file*, e.g. `file tmp.txt` can help provide some information. *read.table()* in R takes arguments *fileEncoding* and *encoding* that allow one to specify the encoding as one reads text in. The UNIX utility *iconv* and the R function *iconv()* can help with conversions.

In US installations of R, the default encoding is UTF-8; note that various types of information are interpreted in US English with the encoding UTF-8:

```
Sys.getlocale()

## [1] "LC_CTYPE=en_US.UTF-8;LC_NUMERIC=C;LC_TIME=en_US.UTF-8;LC_COLLATE=en_
```

With strings already in R, you can convert between encodings with *iconv()*:

```
text <- "_Melhore sua seguran\xe7a_"
Encoding(text)

## [1] "unknown"

Encoding(text) <- "latin1"
text

## [1] "_Melhore sua seguranÃ§a_"

text <- "_Melhore sua seguran\xe7a_"
textUTF8 <- iconv(text, from = "latin1", to = "UTF-8")
Encoding(textUTF8)

## [1] "UTF-8"

textUTF8

## [1] "_Melhore sua seguranÃ§a_"

iconv(text, from = "latin1", to = "ASCII", sub = "???")

## [1] "_Melhore sua seguran???a_"
```

You can also mark a string with an encoding, so R knows how to display it correctly:

```
x <- "fa\xE7ile"
Encoding(x) <- "latin1"
x

## [1] "faÃ§ile"

## playing around...
x <- "\xa1 \xa2 \xa3 \xf1 \xf2"
Encoding(x) <- "latin1"
x

## [1] "Â¡ Â¢ Â£ Ã± ²"
```

An R error message with "multi-byte string" in the message often indicates an encoding issue. In particular errors often arise when trying to do string manipulations in R on character vectors for which the encoding is not properly set. Here's an example with some Internet logging data that we used a few years ago in class in a problem set and which caused some problems.

```
load('../data/IPs.RData') # loads in an object named 'text'
tmp <- substring(text, 1, 15)

## Error in substring(text, 1, 15):  invalid multibyte string at '<bf>a7lw8
[128.32.244.179] by ncpc-email with ESMTP
## (SMTPD32-7.04) id A06E24A0116; Mon, 10 Jun 2002 11:43:42 +0800'

## the issue occurs with the 6402th element (found by trial and error):
tmp <- substring(text[1:6401],1,15)
tmp <- substring(text[1:6402],1,15)

## Error in substring(text[1:6402], 1, 15):  invalid multibyte string
at '<bf>a7lw8<6c>z2nX,%@ [128.32.244.179] by ncpc-email with ESMTP
## (SMTPD32-7.04) id A06E24A0116; Mon, 10 Jun 2002 11:43:42 +0800'

text[6402] # note the Latin-1 character

## [1] "from 5#c\xbfa7lw8lz2nX,%@ [128.32.244.179] by ncpc-email with ESMTP

table(Encoding(text))
```

```
##
## unknown
##    6936

## Option 1
Encoding(text) <- "latin1"
tmp <- substring(text, 1, 15)
tmp[6402]

## [1] "from 5#cÂ¿a7lw8l"

## Option 2
load('../data/IPs.RData') # loads in an object named 'text'
tmp <- substring(text, 1, 15)

## Error in substring(text, 1, 15):  invalid multibyte string at '<bf>a7lw8
[128.32.244.179] by ncpc-email with ESMTP
## (SMTPD32-7.04) id A06E24A0116; Mon, 10 Jun 2002 11:43:42 +0800'

text <- iconv(text, from = "latin1", to = "UTF-8")
tmp <- substring(text, 1, 15)
```

5 Output from R

5.1 Writing output to files

Functions for text output are generally analogous to those for input. *write.table()*, *write.csv()*, and *writeLines()* are analogs of *read.table()*, *read.csv()*, and *readLines()*. *write_csv()* is the *readr* version of write.csv. *write()* can be used to write a matrix to a file, specifying the number of columns desired. *cat()* can be used when you want fine control of the format of what is written out and allows for outputting to a connection (e.g., a file).

toJSON() in the *jsonlite* package will output R objects as JSON. One use of JSON as output from R would be to *serialize* the information in an R object such that it could be read into another program.

And of course you can always save to an R data file using *save.image()* (to save all the objects in the workspace or *save()* to save only some objects. Happily this is platform-independent so can be used to transfer R objects between different OS.

5.2 Formatting output

cat() is a good choice for printing a message to the screen, often better than *print()*, which is an object-oriented method. You generally won't have control over how the output of a *print()* statement is actually printed.

```
val <- 1.5
cat('My value is ', val, '.\n', sep = '')

## My value is 1.5.

print(paste('My value is ', val, '.', sep = ''))

## [1] "My value is 1.5."
```

We can do more to control formatting with *cat()*:

```
## input
x <- 7
n <- 5
## display powers
cat("Powers of", x, "\n")

## Powers of 7

cat("exponent   result\n\n")

## exponent   result

result <- 1
for (i in 1:n) {
        result <- result * x
        cat(format(i, width = 8), format(result, width = 10),
            "\n", sep = "")
}

##        1         7
##        2        49
##        3       343
##        4      2401
##        5     16807
```

```
x <- 7
n <- 5
## display powers
cat("Powers of", x, "\n")

## Powers of 7

cat("exponent result\n\n")

## exponent result

result <- 1
for (i in 1:n) {
        result <- result * x
        cat(i, '\t', result, '\n', sep = '')
}

## 1 7
## 2 49
## 3 343
## 4 2401
## 5 16807
```

One thing to be aware of when writing out numerical data is how many digits are included. For example, the default with *write()* and *cat()* is the number of digits that R displays to the screen, controlled by `options()$digits`. But note that `options()$digits` seems to have some variability in behavior across operating systems. If you want finer control, use *sprintf()*, e.g., to print out print out temperatures as reals ("*f*"=floating points) with four decimal places and nine total character positions, followed by a C for Celsius:

```
temps <- c(12.5, 37.234324, 1342434324.79997234, 2.3456e-6, 1e10)
sprintf("%9.4f C", temps)

## [1] "  12.5000 C"         "  37.2343 C"
## [3] "1342434324.8000 C"   "   0.0000 C"
## [5] "10000000000.0000 C"

city <- "Boston"
sprintf("The temperature in %s was %.4f C.", city, temps[1])
```

```
## [1] "The temperature in Boston was 12.5000 C."
```

```
sprintf("The temperature in %s was %9.4f C.", city, temps[1])
```

```
## [1] "The temperature in Boston was   12.5000 C."
```

Note, to change the number of digits printed to the screen, do `options(digits = 5)` or specify as an argument to *print()* or use *sprintf()*.