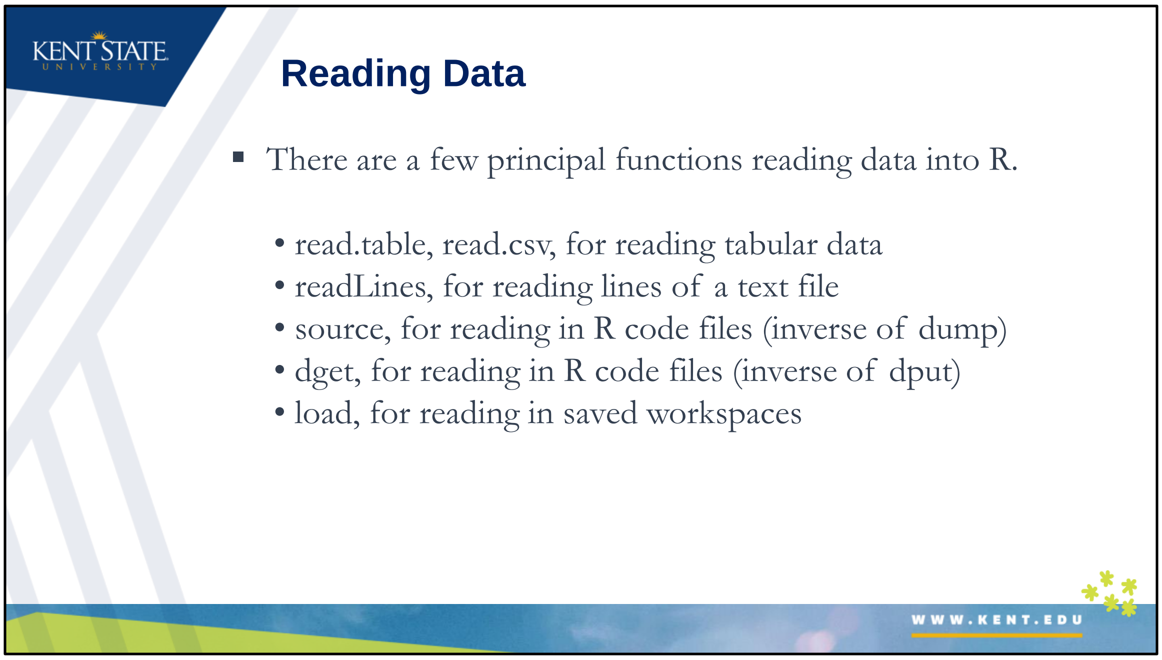
KENT STATE UNIVERSITY

Data Input / Output Operations in R

WWW.KENT.EDU

All statistical work begins with data, and most data is stored inside files and databases. This presentation introduces input and output operations in R and provides examples.

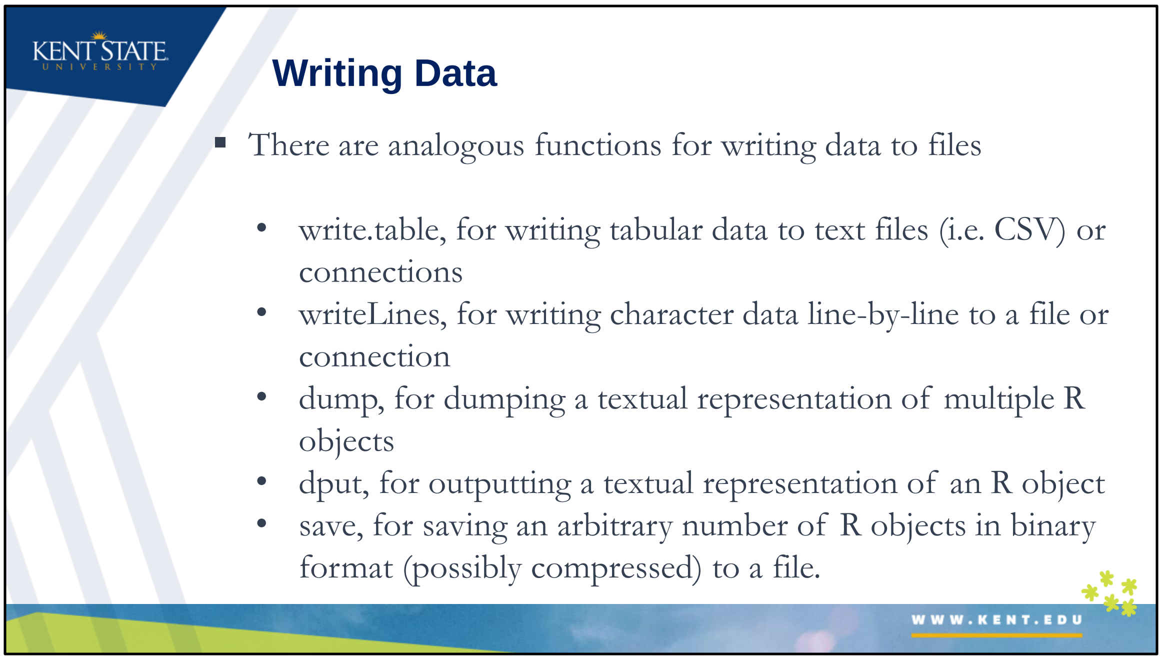


Reading Data

- There are a few principal functions reading data into R.
  - read.table, read.csv, for reading tabular 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

WWW.KENT.EDU

Depending on the type of data base or file, there are different ways to read data into R. read.table and read.csv are two common ways for reading tabular data. If you want to read lines of text files you can use the read line function. Source and load functions are also common to read the R dump files and save workspaces into R. We will examine some of these functions in more detail later on.

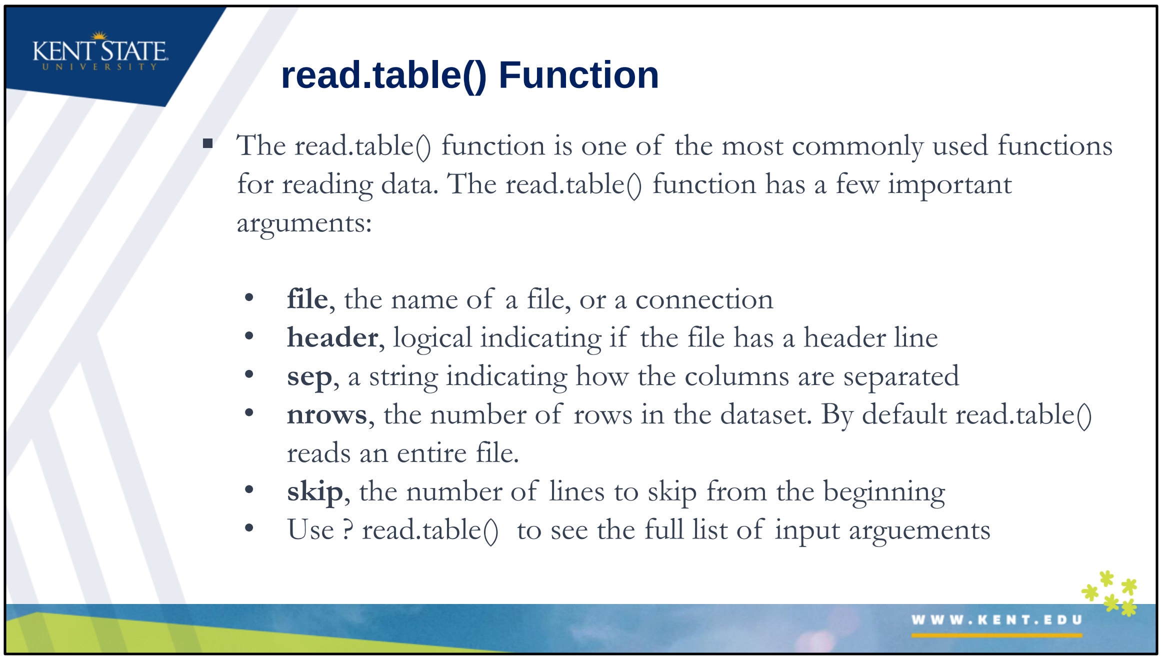


Writing Data

- There are analogous functions for writing data to files
  - write.table, for writing tabular data to text files (i.e. CSV) or connections
  - writeLines, for writing character data line-by-line to a file or connection
  - 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 (possibly compressed) to a file.

WWW.KENT.EDU

There are also a number of functions that can help write data into files. We have write.table, write.csv, write lines dump, dput, and save. Similar to reading the files, here we have write.table, write.csv, for writing tabular data into the files, write lines for writing characters of data line by line and dump in order to dump in order to dump in the text world presentation of multiple R objects in the file and save for saving arbitrary number of objects into binary format.

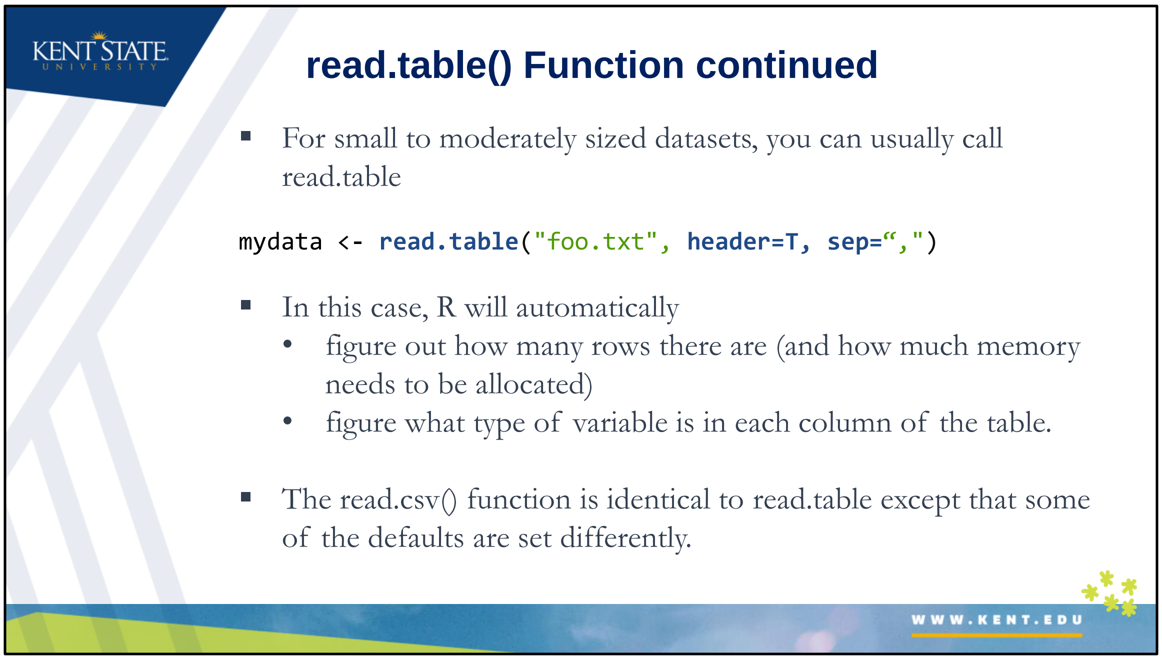


read.table() Function

- The read.table() function is one of the most commonly used functions for reading data. The read.table() function has a few important arguments:
  - **file**, the name of a file, or a connection
  - **header**, logical indicating if the file has a header line
  - **sep**, a string indicating how the columns are separated
  - **nrows**, the number of rows in the dataset. By default read.table() reads an entire file.
  - **skip**, the number of lines to skip from the beginning
  - Use ? read.table() to see the full list of input arguements

WWW.KENT.EDU

Let's start with the read table function. This function is considered one of the most common function for reading the data. The function takes a number of variables. Some of the most important ones are listed here. The first variable is file which refers to the name of a file or connection; we will talk about connections later on in this presentation. Header is a logical flag which indicates where the file has a header or not. If the header is set to true, then the first line of the file is interpreted as the name of the columns. Sep is a string that indicates how the columns are separated, so in different files columns could be separated using tab, space, comma, etc. The number of rows or end rows defines the number of rows of the data files that needs to be read. By default the function reads the entire file. And skip defines the number of lines to skip from the beginning. So, for example, if you want to start reading a file from line number ten, you can set the skip to nine, so the first nine line of the file would be skipped. I also strongly encourage you to use the help function of the R. if you print question mark and then provide the name of any function in the R terminal, all the provided documentation regarding that function. This includes every single input to the file, all the default values and the output values and some examples towards the end.



read.table() Function continued

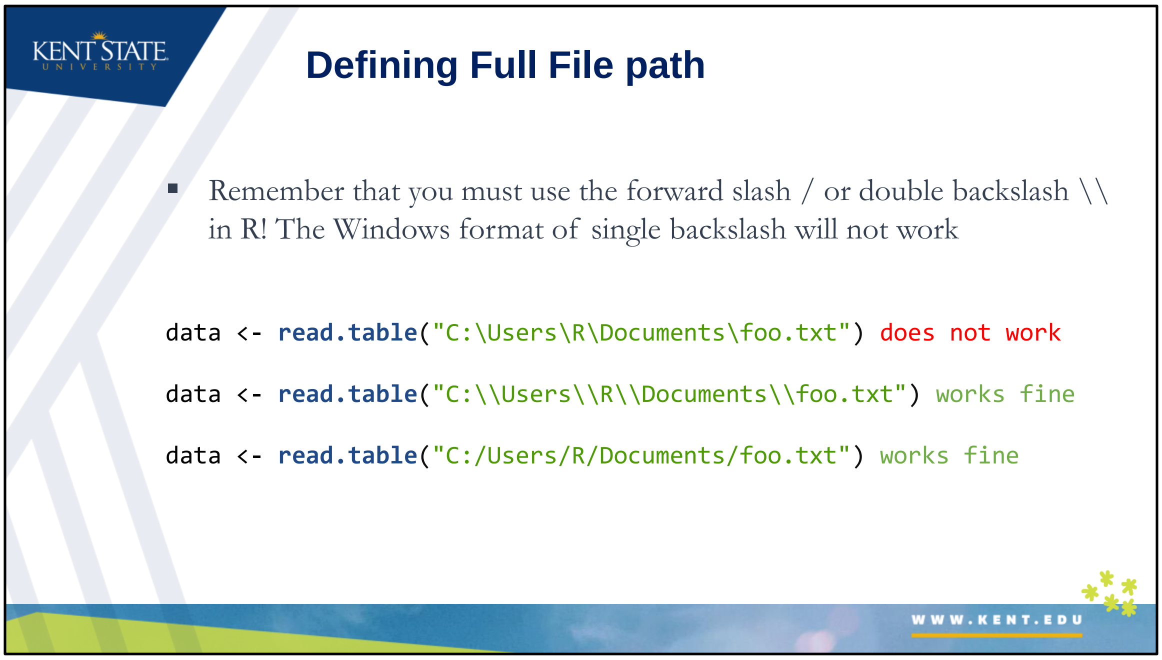
- For small to moderately sized datasets, you can usually call read.table

```
mydata <- read.table("foo.txt", header=T, sep=“,")
```

- In this case, R will automatically
  - figure out how many rows there are (and how much memory needs to be allocated)
  - figure what type of variable is in each column of the table.
- The read.csv() function is identical to read.table except that some of the defaults are set differently.

WWW.KENT.EDU

Read table function works perfectly fine for small or moderate sized files. Here is a simple example of how you can use the function. So in this case we have my data, which is a data frame, containing the data that is read from the foo.text file. So, foo.text is the name of the file, and then we have the file header as true which means that the first line of the file contains the name of the variables, and then columns are separated by commas in this case. From using the function, R automatically figures the number of rules that need to be read and therefore allocates the necessary amount of memory. It also figures the type of variable in each column. Normally, number columns are stored as numeric, and string variables are stored as factors. The read.csv is very similar to read.table, except that some of the default values are set slightly differently.

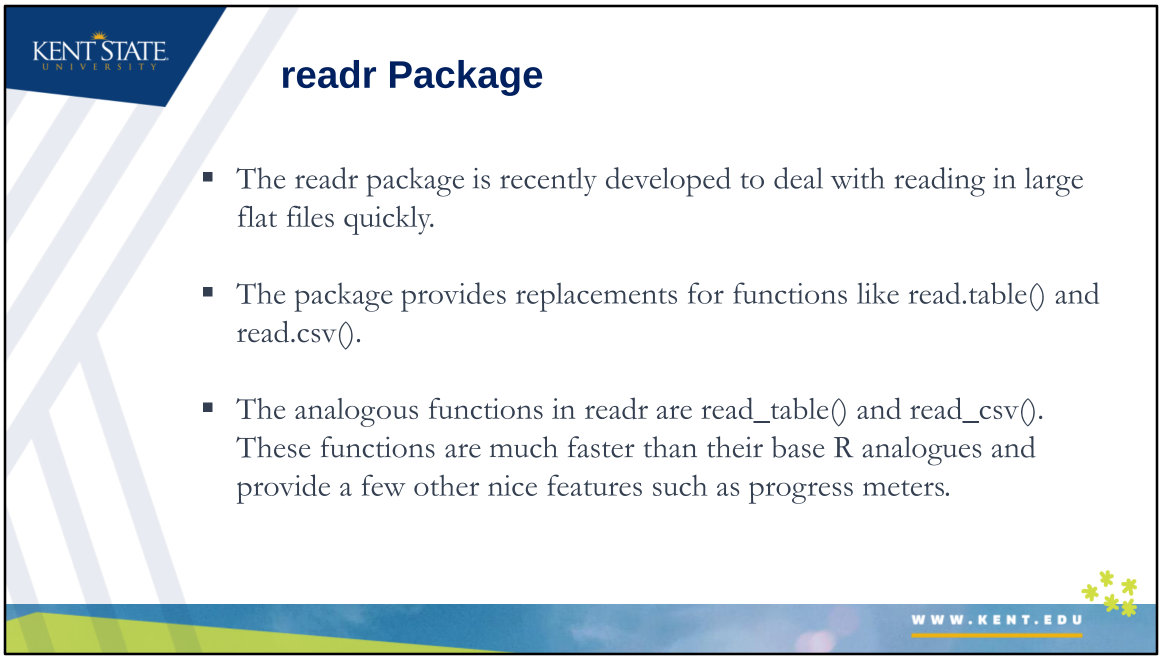


Defining Full File path

- Remember that you must use the forward slash / or double backslash \\ in R! The Windows format of single backslash will not work

```
data <- read.table("C:\Users\R\Documents\foo.txt") does not work

data <- read.table("C:\\Users\\R\\Documents\\foo.txt") works fine

data <- read.table("C:/Users/R/Documents/foo.txt") works fine
```

Here is an important note about defining the full file path. The number you must use is the forward slash or double backslash in R in order to define the path for the file. The Windows format of single backslash will not work for R. For example, if you want to read the file called foo.text which is located in user/R/documents the first line of the code would not work. We can either change the single backslashes to double backslashes, or change them to forward slashes.

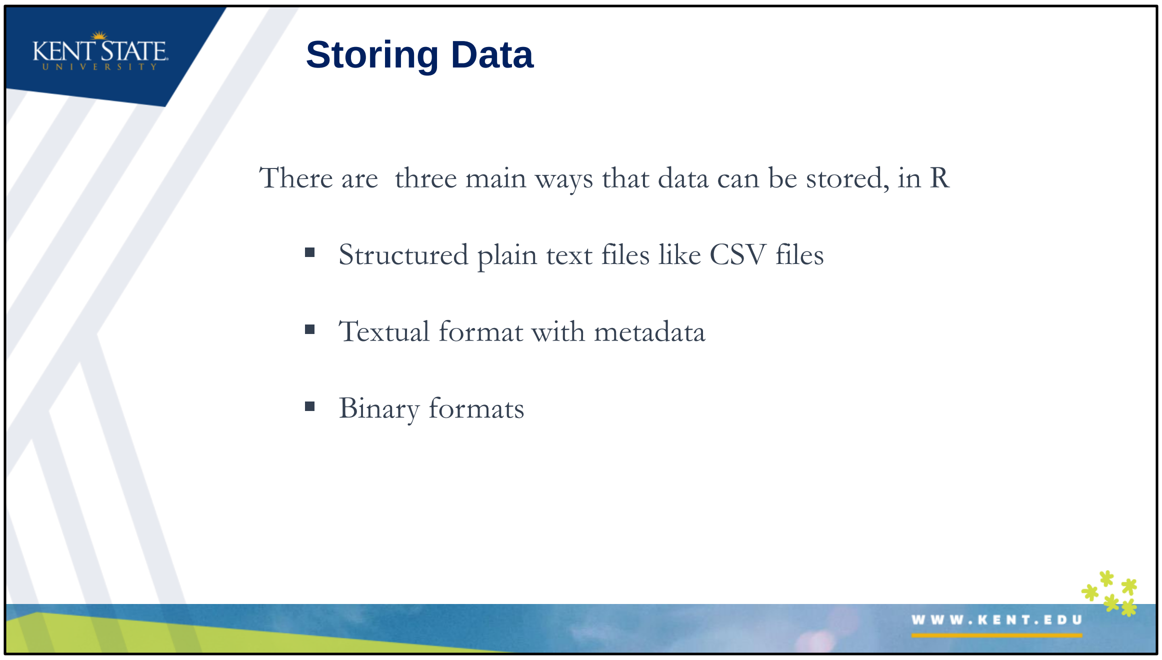


readr Package

- The readr package is recently developed to deal with reading in large flat files quickly.
- The package provides replacements for functions like read.table() and read.csv().
- The analogous functions in readr are read_table() and read_csv(). These functions are much faster than their base R analogues and provide a few other nice features such as progress meters.

WWW.KENT.EDU

The readr package is recently developed to help reading large files faster and more efficiently. The package provides some functions that can replace read.table and read.csv. more specifically read_table and read_csv are equivalent functions that can work exactly the same way as read.table and read.csv reworking, but they're much faster and much more efficient. If you are working with the larger files, it is strongly recommended that you use these functions instead of the base functions.



Storing Data

There are three main ways that data can be stored, in R

- Structured plain text files like CSV files
- Textual format with metadata
- Binary formats

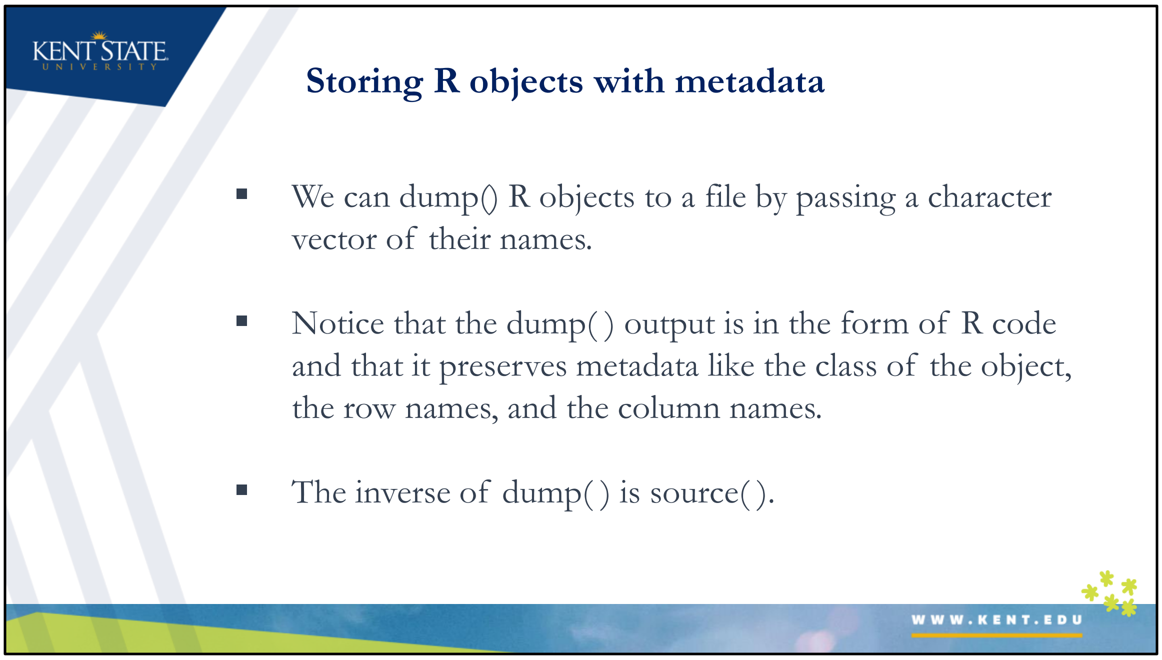
WWW.KENT.EDU

There are three main ways that data can be stored, in R : plaint text files, text files with metadata, and finally binary files.

Textual formats can work much better with version control programs like git which can only track changes meaningfully in text files. In addition, textual formats can be more robust;
For example, if there is corruption somewhere in the file, it is easier to fix the problem because one can just
open the file in an editor and look at it

The complement to the textual format is the binary format, which is sometimes necessary to use
for efficiency purposes, or because there's just no useful way to represent data in a textual manner.
Also, with numeric data, one can often lose precision when converting to and from a textual format,
so it's better to stick with a binary format.

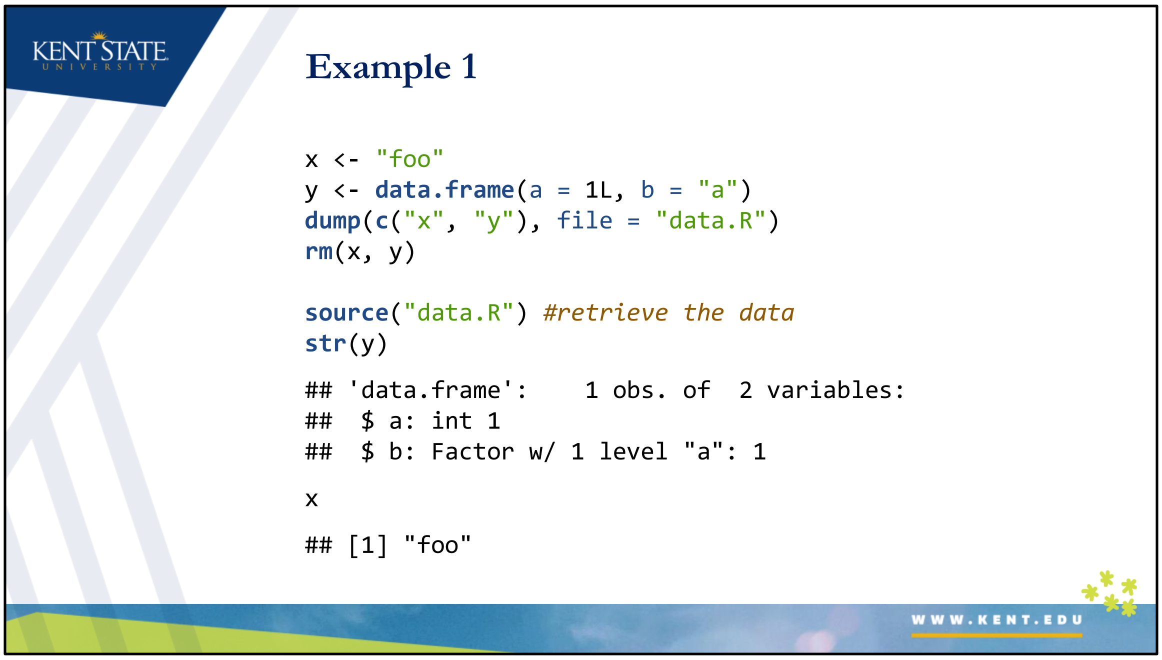
One can create a more descriptive representation of an R object by using the dput()
or dump()
functions. The dump() and dput() functions are useful because the resulting textual
format is editable,
and in the case of corruption, potentially recoverable. However, unlike writing out a
table or CSV file,
dump() and dput() preserve the *metadata*, so that another user doesn’t
have to specify it all over again.



Storing R objects with metadata

- We can dump() R objects to a file by passing a character vector of their names.
- Notice that the dump( ) output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names.
- The inverse of dump( ) is source( ).

As we said before, we can dump R function to a file by passing the character vector of their names. In this case dump will create output that is in the form of the R code. So, in other words, it's a textual file which includes also some metta data, like class of the objects, row names and the column names. The inverse of the dump function is the source function. So, once we have dumped a content in a file, we can use the source function to retract the data back.



Example 1

```
x <- "foo"
y <- data.frame(a = 1L, b = "a")
dump(c("x", "y"), file = "data.R")
rm(x, y)

source("data.R") #retrieve the data
str(y)
```

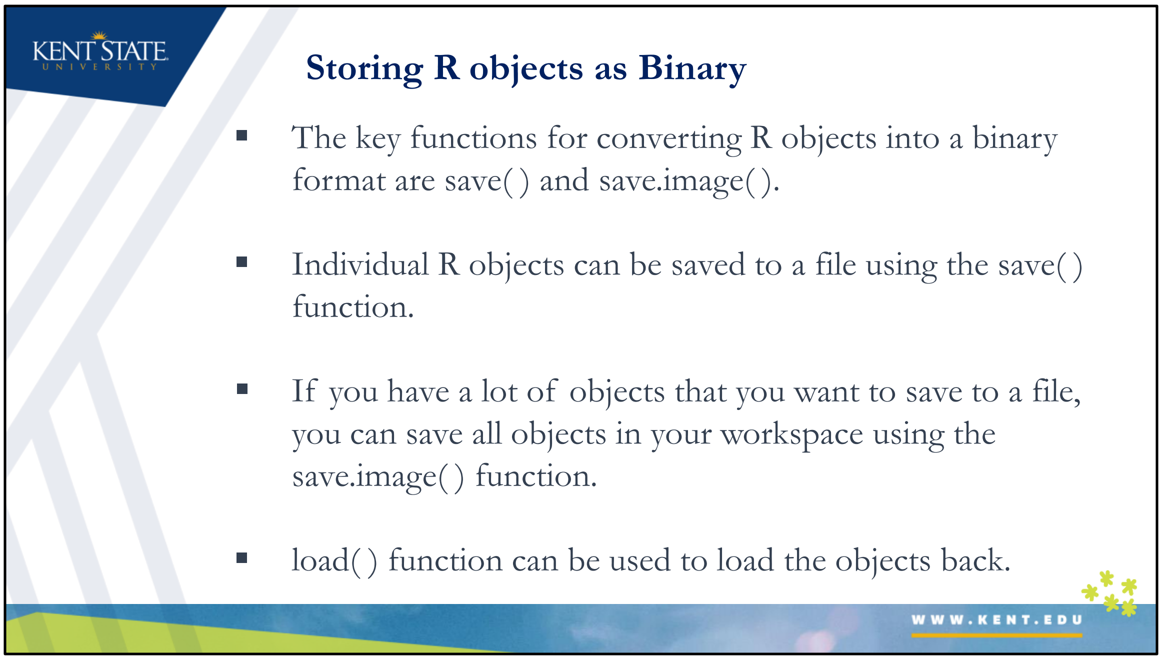
```
## 'data.frame':    1 obs. of  2 variables:
##  $ a: int 1
##  $ b: Factor w/ 1 level "a": 1
```

```
x
```

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

WWW.KENT.EDU

Let’s have a look at a quick example. In this example, we create a variable x, which contains strain foo, we also have a data frame y, which consists of two variables, a, which is an integer, with the value of one, and b, which is a factor and contains a string factor of a. In this case, we want to dump x and y as two variables, into a file called data.r. After we dump these files, we remove variables x and y. Now we can use the source function to retrieve back the data. So, we have source data.r. Now, if you look at this structure of y, we can see that now y is considered as a data frame, which has two variables, a and b; a is defined as an integer, b is a factor with one level a. in this case, because the meta data has been saved together with the data, R already knows that a is an integer value and not a numeric value. If we were saving this using the plain text, the variable a in the data frame y, could have been interpreted as a numeric. We can also see the value of x retrieved as foo.

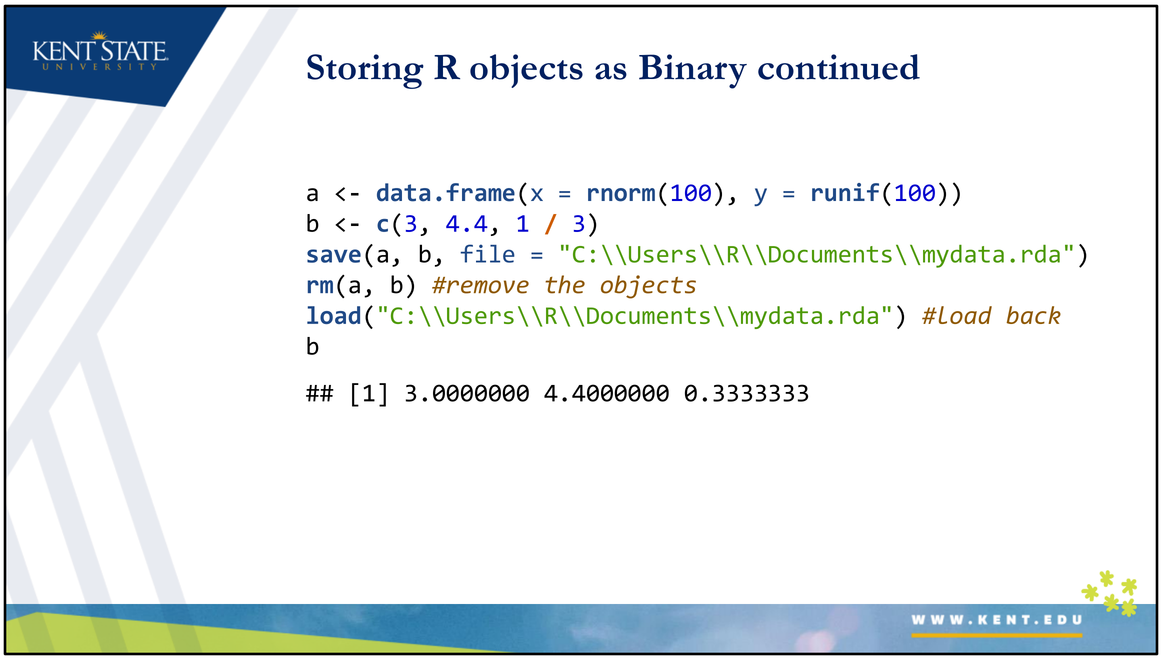


Storing R objects as Binary

- The key functions for converting R objects into a binary format are save( ) and save.image( ).
- Individual R objects can be saved to a file using the save( ) function.
- If you have a lot of objects that you want to save to a file, you can save all objects in your workspace using the save.image( ) function.
- load( ) function can be used to load the objects back.

WWW.KENT.EDU

For storing R objects in binary, the easiest functions to use are save and save.image. in fact, if you want to save individual R objects, save would be the best function to use. However, if you want to save the entire work space of R, you can use the save.image function. Load is a function that can be used to load binary R objects.



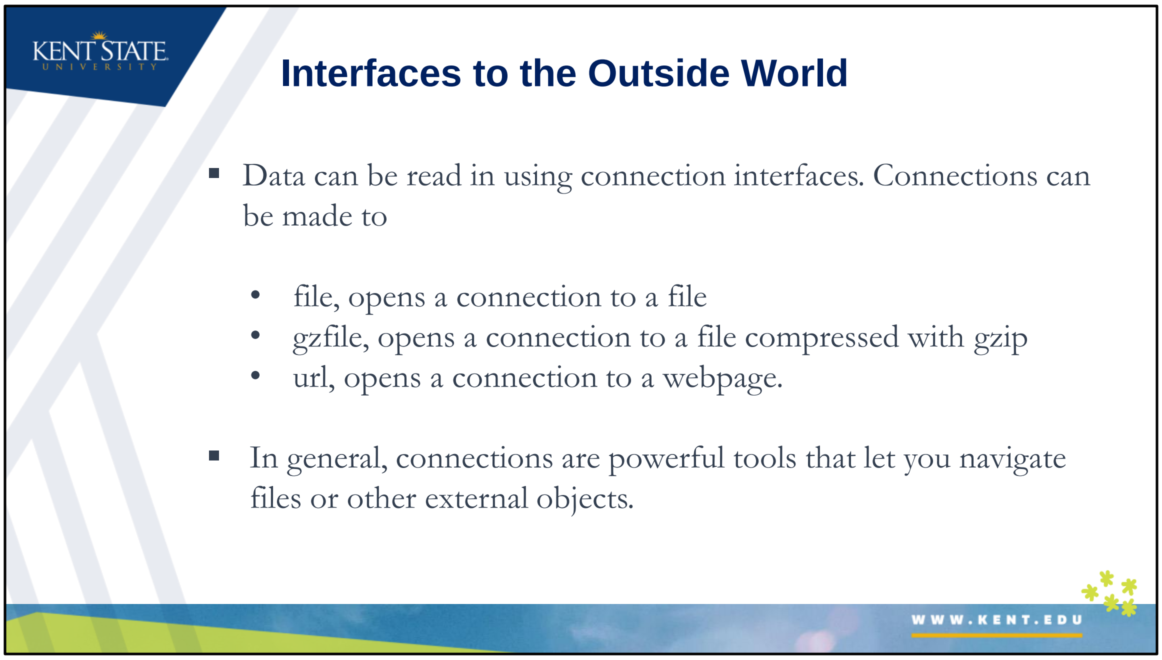
Storing R objects as Binary continued

```
a <- data.frame(x = rnorm(100), y = runif(100))
b <- c(3, 4.4, 1 / 3)
save(a, b, file = "C:\\Users\\R\\Documents\\mydata.rda")
rm(a, b) #remove the objects
load("C:\\Users\\R\\Documents\\mydata.rda") #Load back
b
```

```
## [1] 3.0000000 4.4000000 0.3333333
```

WWW.KENT.EDU

Another quick example is shown in here. So, in this case, we have a data frame called a, which contains two variables x and y. We also have a variable b, which is a vector of integer values. In this case, we save a and b into a file called mydata.rda. Note, that in this example we, define the foo pass. You soul also know we have used double backslashes in here. Now that we have saved these two objects, use the RM function to remove these two objects. We use the load functions again to load the data back. So, once the load command is executed, we can easily type b and we can see values of b printed out.

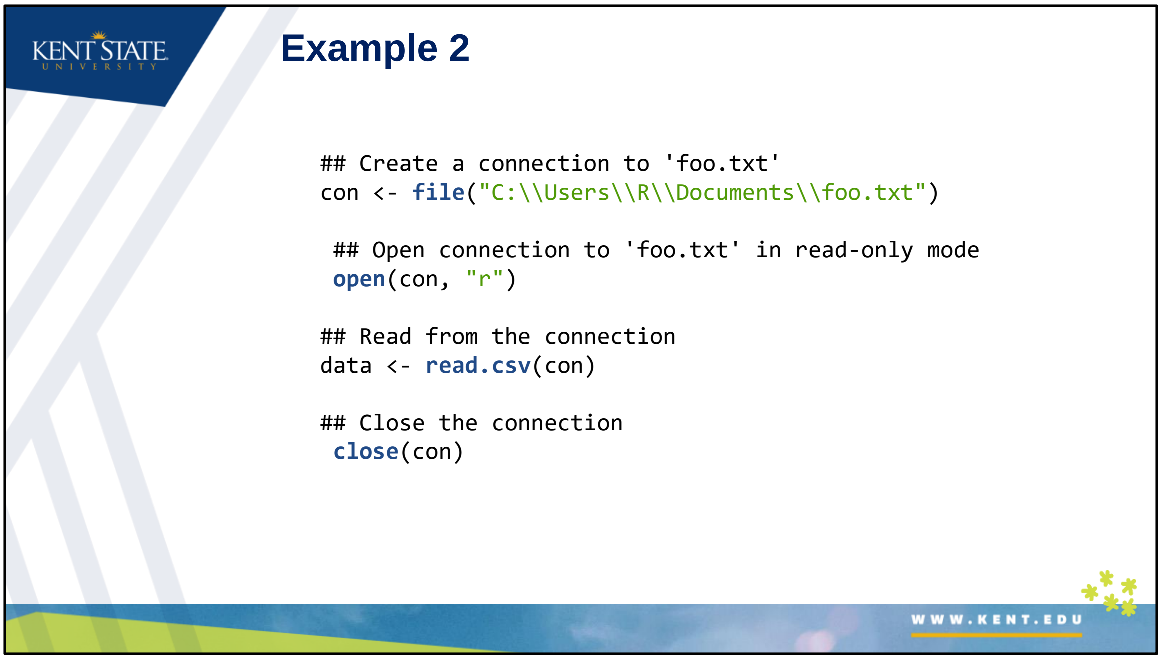


Interfaces to the Outside World

- Data can be read in using connection interfaces. Connections can be made to
  - file, opens a connection to a file
  - gzfile, opens a connection to a file compressed with gzip
  - url, opens a connection to a webpage.
- In general, connections are powerful tools that let you navigate files or other external objects.

WWW.KENT.EDU

Data can also be read in using connection interfaces where connections can be made to data files, zipped files or remote URLs.
Connections can be thought of as a translator that lets you talk to objects that are outside of R.
Connections allow R functions to talk to all these different external objects without you having to
write custom code for each object.

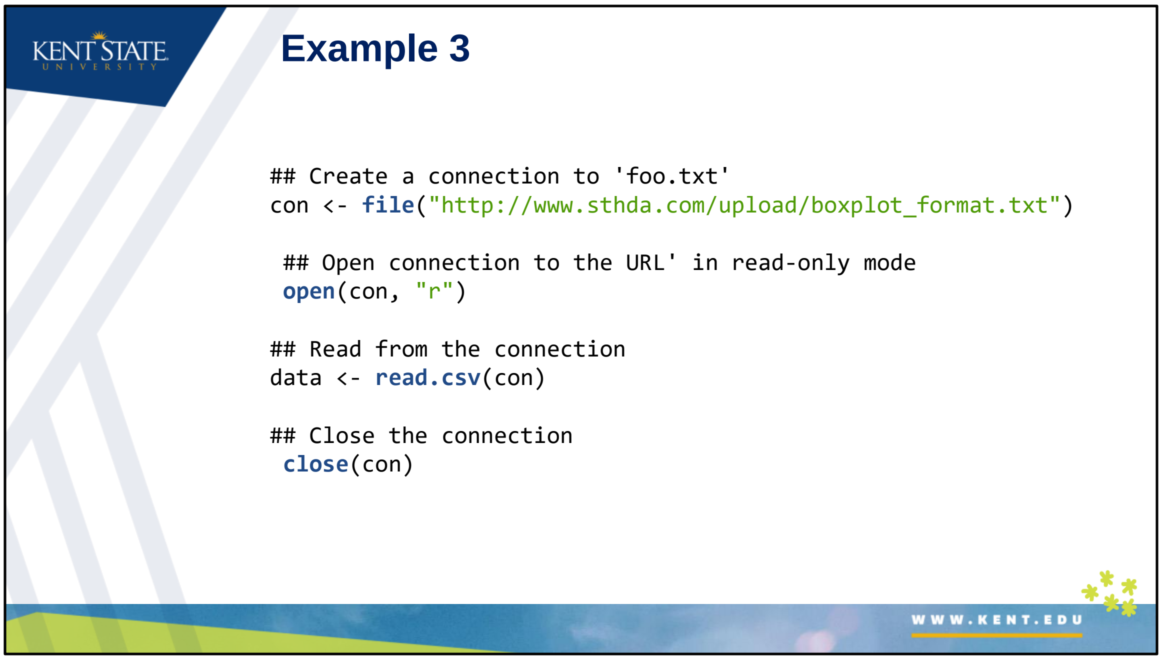


Example 2

```
## Create a connection to 'foo.txt'
con <- file("C:\\Users\\R\\Documents\\foo.txt")

 ## Open connection to 'foo.txt' in read-only mode
 open(con, "r")

## Read from the connection
data <- read.csv(con)

## Close the connection
 close(con)
```

WWW.KENT.EDU

Here is the simple example of how you can use connections to read a file. So, in this case instead of reading the file directly, we create a connection object. So, in that case we first create a connection to a file, where the file is foo text and a foo pass is provided, and then we need to open a connection to a foo text file in read only mode. So, we have open, connection and R. When we open the connection and specify the type of connection as R, it means that this connection is a read only connection. This means that this connection allows us to read from the file, but we cannot write into the file using this connection. Now we use the read.csv file as we have used before, but instead of providing the file name, we pass the connection object. Once the file is read, we can close the connection using the close function.



Example 3

```
## Create a connection to 'foo.txt'
con <- file("http://www.sthda.com/upload/boxplot_format.txt")

 ## Open connection to the URL' in read-only mode
 open(con, "r")

## Read from the connection
data <- read.csv(con)

## Close the connection
 close(con)
```

WWW.KENT.EDU

Here is another very similar example, where instead we are reading the file from an online URL. So, in this case, instead of defining the foo password file, we define a URL where the file is located. After the connection object is created, we can open the connection, read the file and close the connection.