DATA SOCIETY®

Intro to R programming - day 3

"One should look for what is and not what he thinks should be."
-Albert Einstein.

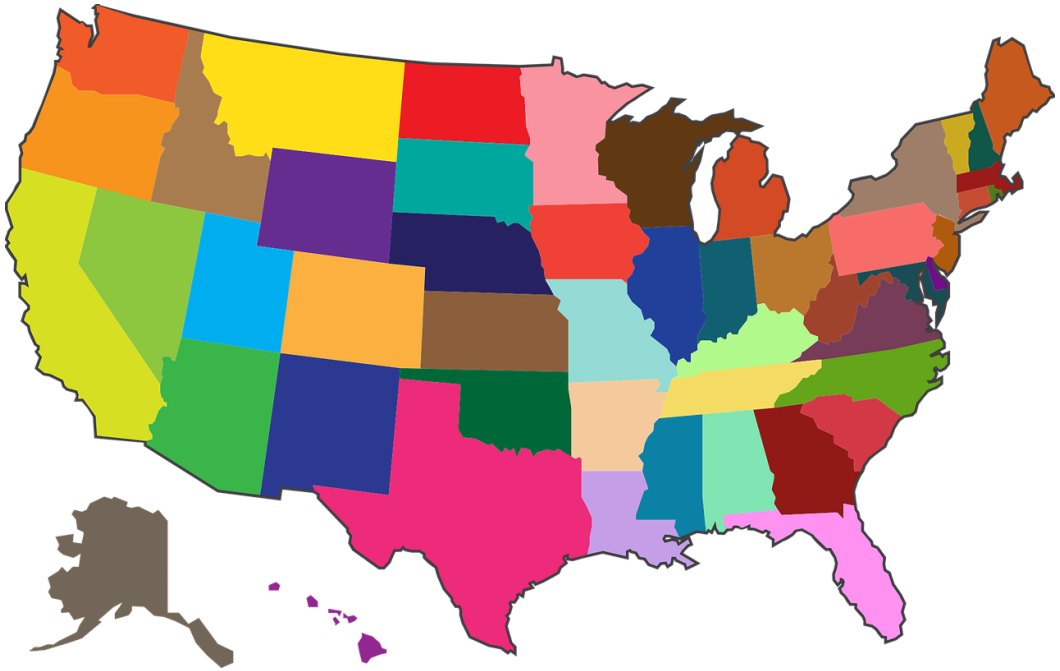
Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	
Demonstrate working with the random number generator	
Explain apply family of functions as an alternative to for loops	
Use lapply on dataset	
Discuss sapply and use sapply with dataframe	
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Datasets in R: `state.x77` data

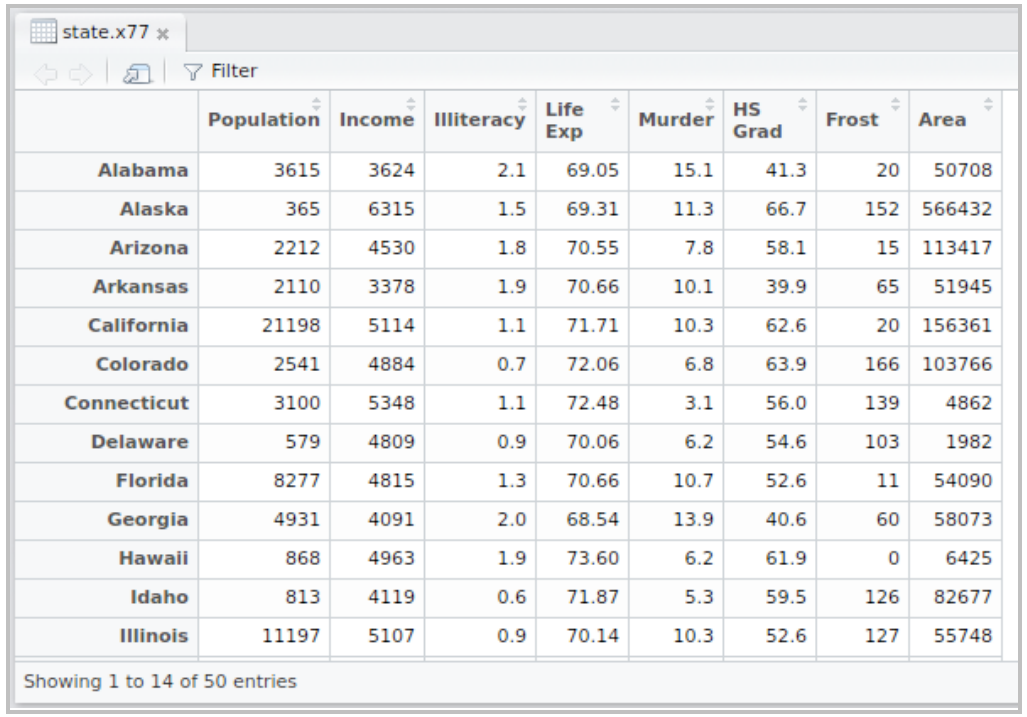
We are going to make use of some datasets that come pre-loaded with R. These datasets have some data related to the 50 US states and include:

- `state.x77`
- `state.abb`
- `state.area`
- `state.name`
- `state.region`
- `state.center`
- `state.division`



Datasets in R: `state.x77` data

```
# Let's take a look at `state.x77` dataset.
View(state.x77)
```



state.x77

Filter

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945
California	21198	5114	1.1	71.71	10.3	62.6	20	156361
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766
Connecticut	3100	5348	1.1	72.48	3.1	56.0	139	4862
Delaware	579	4809	0.9	70.06	6.2	54.6	103	1982
Florida	8277	4815	1.3	70.66	10.7	52.6	11	54090
Georgia	4931	4091	2.0	68.54	13.9	40.6	60	58073
Hawaii	868	4963	1.9	73.60	6.2	61.9	0	6425
Idaho	813	4119	0.6	71.87	5.3	59.5	126	82677
Illinois	11197	5107	0.9	70.14	10.3	52.6	127	55748

Showing 1 to 14 of 50 entries

- The data is in tabular format, but we are not sure which kind, since R supports matrices, tables and dataframes and each of them has different properties
- To quickly check the type (or `class`, as it is called in R community), we can use the `class` function

```
class(state.x77)
```

```
[1] "matrix"
```

Checking the dimension of our dataset

```
# This dataset is of type `matrix`. We don't want to modify the original dataset,
# so let's set this dataset to a variable, so that we can manipulate it freely.
state_data = state.x77

# The dataset contains 50 rows (i.e. 50 states) and 8 columns.
# It's easy to check the dimensions of any object in R with a simple `dim` function.
dim(state_data)
```

```
[1] 50  8
```

```
# Since matrix is a 2-dimensional object we get a vector with 2 entries:
# 1. The first one corresponds to the number of rows
dim(state_data)[1]
```

```
[1] 50
```

```
# 2. The second tells us how many columns we have
dim(state_data)[2]
```

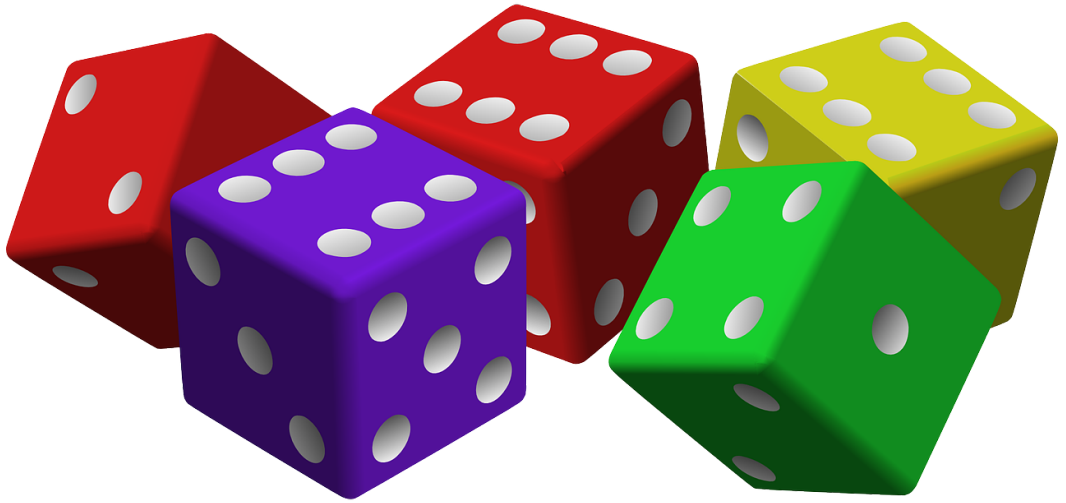
```
[1] 8
```

Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	
Explain apply family of functions as an alternative to for loops	
Use lapply on dataset	
Discuss sapply and use sapply with dataframe	
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Random number generators & sampling in R

- What is **random**?
- Why random numbers?
- Where do we use them?
- How to generate them in R?



Random number generators & sampling in R

- Numbers generated in R: are they truly random?
- Where can we get truly random numbers?
- Why bother?

RANDOM.ORG

Do you own an iOS or Android device? Check out our app!

What's this fuss about *true* randomness?

Perhaps you have wondered how predictable machines like computers can generate randomness. In reality, most random numbers used in computer programs are *pseudo-random*, which means they are generated in a predictable fashion using a mathematical formula. This is fine for many purposes, but it may not be random in the way you expect if you're used to dice rolls and lottery drawings.

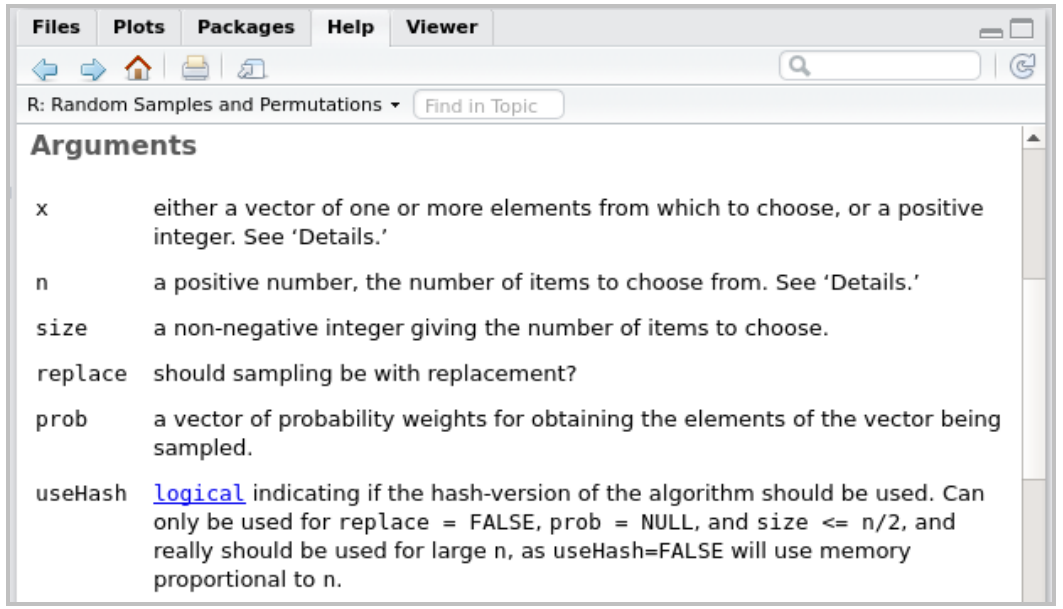
RANDOM.ORG offers *true* random numbers to anyone on the Internet. The randomness comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs. People use RANDOM.ORG for holding drawings, lotteries and sweepstakes, to drive online games, for scientific applications and for art and music. The service has existed since 1998 and was built by Dr Mads Haahr of the School of Computer Science and Statistics at Trinity College, Dublin in Ireland. Today, RANDOM.ORG is operated by Randomness and Integrity Services Ltd.

As of today, RANDOM.ORG has generated 1.64 trillion random bits for the Internet community.

Generating a random sample of items

- One of the easiest and most often used random number generating functions in R is `sample`

```
?sample

sample(x,     #<- a vector of items
              #   or a positive integer
              #   from which to choose
       size,  #<- a non-negative integer
              #   giving the number
              #   of items to generate
       ...)   #<- other arguments
```



Generating states in random

```
# Generate 10 numbers from a
# range of numbers from 1 to 100.
sample(1:100,
       size = 10)
```

```
 [1] 86 17 55 30 93 60 64 95 38 42
```

```
# Generate 5 names of states
# from a vector of 50 states.
sample(state.name,
       size = 5)
```

```
[1] "Wyoming"     "Vermont"     "Kentucky"
"Connecticut" "Minnesota"
```

```
# Generate 10 numbers from a
# range of numbers from 1 to 100.
sample(1:100,
       size = 10)
```

```
 [1]  82  44 100  52  55  69  37  79  89   9
```

```
# Generate 5 names of states
# from a vector of 50 states.
sample(state.name,
       size = 5)
```

```
[1] "West Virginia" "California"
"Massachusetts" "Kentucky"
[5] "Nevada"
```

Generating a reproducible random sample

```
# Set seed to any number. We like to
# use `1`, because it's convenient
# and easy to remember.
set.seed(1)

# Generate 10 numbers from a
# range of numbers from 1 to 100.
sample(1:100,
       size = 10)
```

```
 [1] 27 37 57 89 20 86 97 62 58  6
```

```
# Generate 5 names of states
# from a vector of 50 states.
sample(state.name,
       size = 5)
```

```
[1] "Hawaii"         "Florida"        "North
Carolina" "Maine"
[5] "Oklahoma"
```

```
# So long as the seed is the same number,
# the output you get will be the same.
set.seed(1)

# Generate 10 numbers from a
# range of numbers from 1 to 100.
sample(1:100,
       size = 10)
```

```
 [1] 27 37 57 89 20 86 97 62 58  6
```

```
# Generate 5 names of states
# from a vector of 50 states.
sample(state.name,
       size = 5)
```

```
[1] "Hawaii"         "Florida"        "North
Carolina" "Maine"
[5] "Oklahoma"
```

Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	
Use lapply on dataset	
Discuss sapply and use sapply with dataframe	
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Overview of functionals

Function	Use Case	Data Input Type	Data Output Type
lapply	Replaces a `for` loop to make code more modular, clean, reproducible, scalable and efficient.	List, Vector	List
sapply	Replaces a `for` loop to make code more modular, clean, reproducible, scalable and efficient.	List, Vector, Dataframe	Vector
apply	Replaces a `for` loop to make code more modular, clean, reproducible, scalable and efficient.	Matrix, Dataframe	Vector, Matrix

Defining functionals

What we have

- A generic `for` loop
- Difficult to reproduce
- Difficult to read

```
for(i in 1:5){
  out_list[i] =  sqrt(in_list[i])
}
```

What we need

- A functional like `lapply`
- Easy to reproduce
- Easy to read

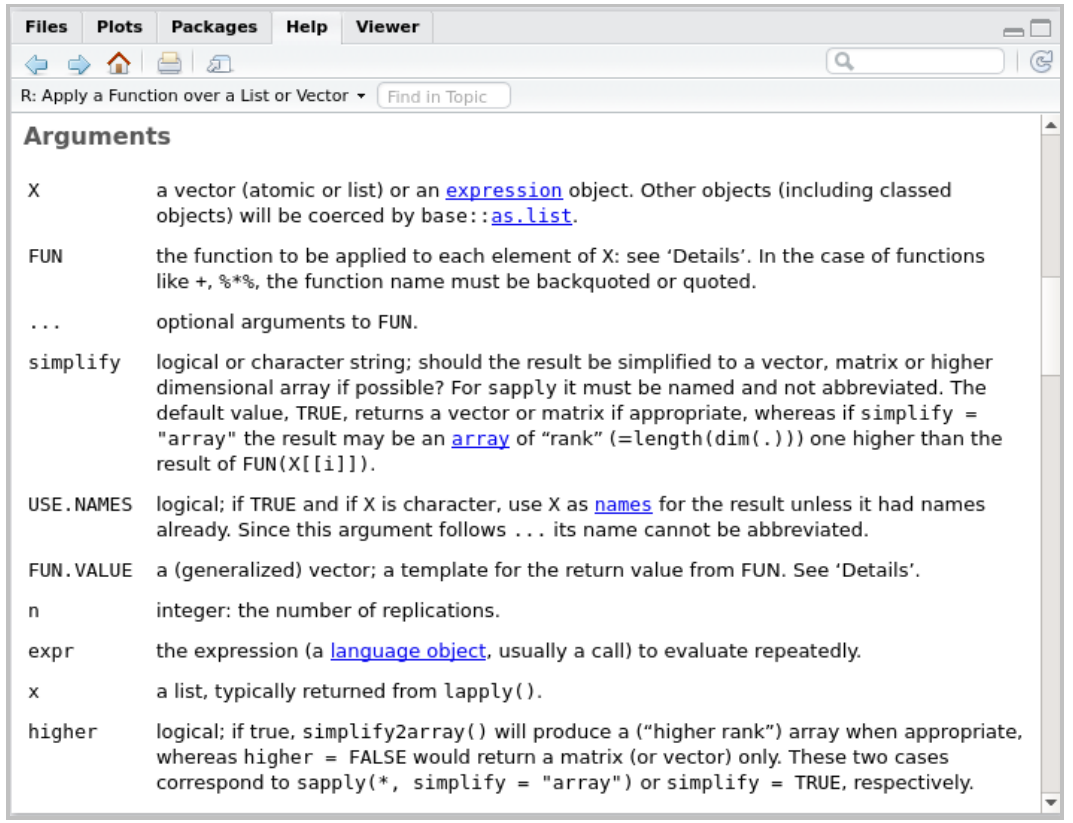
```
out_list = lapply(in_list, sqrt)
```

Introduction to functionals in R

- A **functional** is a function that takes a **vector** , a **list**, or a **matrix** and another **function** *input* and returns a **vector** as *output*:

Understanding lapply

```
?lapply

lapply(X,          #<- a list or a vector
       FUN, ...)   #<- and function
```



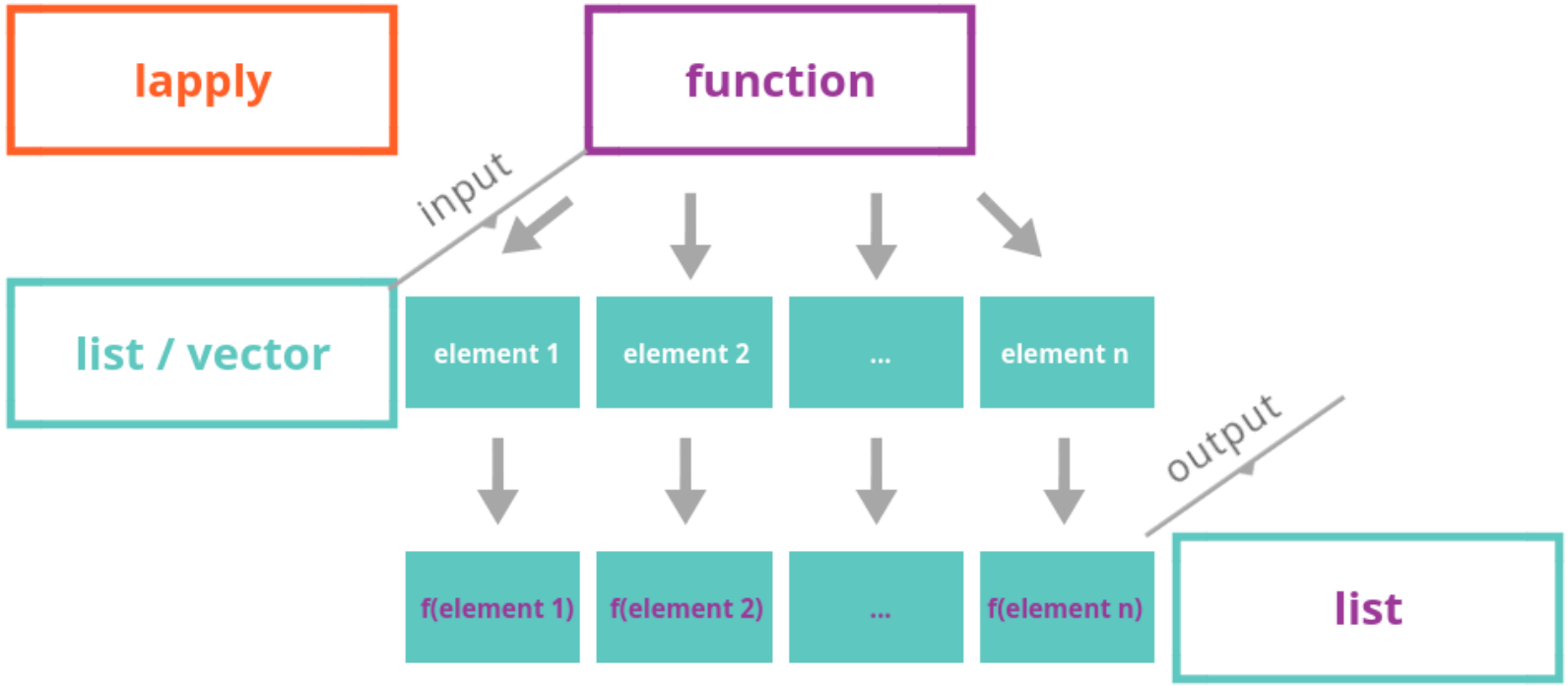
Files | Plots | Packages | Help | Viewer

R: Apply a Function over a List or Vector — Find in Topic

Arguments

X	a vector (atomic or list) or an expression object. Other objects (including classed objects) will be coerced by base::as.list.
FUN	the function to be applied to each element of X: see 'Details'. In the case of functions like +, %*%, the function name must be backquoted or quoted.
...	optional arguments to FUN.
simplify	logical or character string; should the result be simplified to a vector, matrix or higher dimensional array if possible? For sapply it must be named and not abbreviated. The default value, TRUE, returns a vector or matrix if appropriate, whereas if simplify = "array" the result may be an array of "rank" (=length(dim(.))) one higher than the result of FUN(X[[i]]).
USE.NAMES	logical; if TRUE and if X is character, use X as names for the result unless it had names already. Since this argument follows ... its name cannot be abbreviated.
FUN.VALUE	a (generalized) vector; a template for the return value from FUN. See 'Details'.
n	integer: the number of replications.
expr	the expression (a language object, usually a call) to evaluate repeatedly.
x	a list, typically returned from lapply().
higher	logical; if true, simplify2array() will produce a ("higher rank") array when appropriate, whereas higher = FALSE would return a matrix (or vector) only. These two cases correspond to sapply(*, simplify = "array") or simplify = TRUE, respectively.

A quick primer on `lapply` function



Working with `state.region` data

- Let's try out the `lapply` function on a list of character strings first. We will make use of the `state.region` data from the US states datasets

```
# Take a look at the `state.region` vector.
state.region
```

```
 [1] South         West          West          South         West
 [6] West          Northeast     South         South         South
[11] West          West          North Central North Central North Central
[16] North Central South         South         Northeast     South
[21] Northeast     North Central North Central South         North Central
[26] West          North Central West          Northeast     Northeast
[31] West          Northeast     South         North Central North Central
[36] South         West          Northeast     Northeast     South
[41] North Central South         South         West          Northeast
[46] South         West          South         North Central West
Levels: Northeast South North Central West
```

Use `unique` function

- We often have to find distinct entries in a long vector of repeated values. We can do that easily by using the `unique` function

```
# Get a vector of unique regions.
unique_regions = unique(state.region)
unique_regions
```

```
[1] South         West          Northeast     North Central
Levels: Northeast South North Central West
```

Wrap the vector

- Converting between a `vector` and a `list` class is also simple, we just need wrap the vector into `as.list` command

```
# Save the unique regions vector as a list.
unique_regions_list = as.list(unique_regions)
unique_regions_list
```

```
[[1]]
[1] South
Levels: Northeast South North Central West

[[2]]
[1] West
Levels: Northeast South North Central West

[[3]]
[1] Northeast
Levels: Northeast South North Central West

[[4]]
[1] North Central
Levels: Northeast South North Central West
```

Transform a list of US regions with `lapply`

```
# Use `as.character` to convert a single entry of a list
# from `factor` to `character` type.
as.character(unique_regions_list[[1]])
```

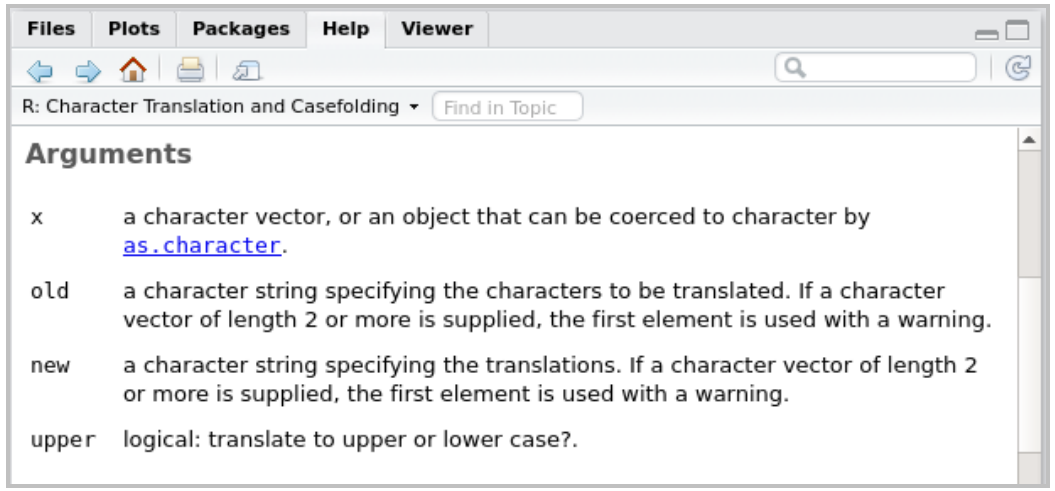
```
[1] "South"
```

```
# By using `lapply`, we can apply `as.character`
# to each element of the list.
unique_regions_list = lapply(unique_regions_list,
                             as.character)
unique_regions_list
```

```
[[1]]
[1] "South"

[[2]]
[1] "West"

[[3]]
[1] "Northeast"

[[4]]
[1] "North Central"
```

Converting character strings to upper case

```
?toupper

toupper(x,    #<- char. string to be converted
        ...) #<- other arguments
```




```
# Convert one of the region
# names to upper case.
toupper(unique_regions_list[[1]])
```

```
[1] "SOUTH"
```

Transform a list of US regions with `lapply`

```
# With the help of our `lapply` function, it is very simple to convert the entire list
# of character strings to upper case.
upper_case_list = lapply(unique_regions_list,  #<- a list that we need to fix
                         toupper)              #<- a function to apply to each entry of list

# Take a look at the output.
upper_case_list
```

```
[[1]]
[1] "SOUTH"

[[2]]
[1] "WEST"

[[3]]
[1] "NORTHEAST"

[[4]]
[1] "NORTH CENTRAL"
```

- Isn't that an easier, more concise and clear way to iterate over a list than your regular `for` loop? The nicest aspect is that you can use virtually *any* function on a list that contains elements of *any* class!

Transform a list of numbers with `lapply`

- Let's make a sample list with some numbers and try our luck with `lapply` on it

```
# Let's make a list with some negative and some positive numbers.
numeric_list = list(first = -1,
                    second = 2,
                    third = 3,
                    fourth = -4)
numeric_list
```

```
$first
[1] -1

$second
[1] 2

$third
[1] 3

$fourth
[1] -4
```

Transform a list of numbers with `lapply`

```
# With this single-liner, we can convert them to
# their absolute value equivalent!
abs_value_list = lapply(numeric_list, abs)
abs_value_list
```

```
$first
[1] 1

$second
[1] 2

$third
[1] 3

$fourth
[1] 4
```

Knowledge check 1

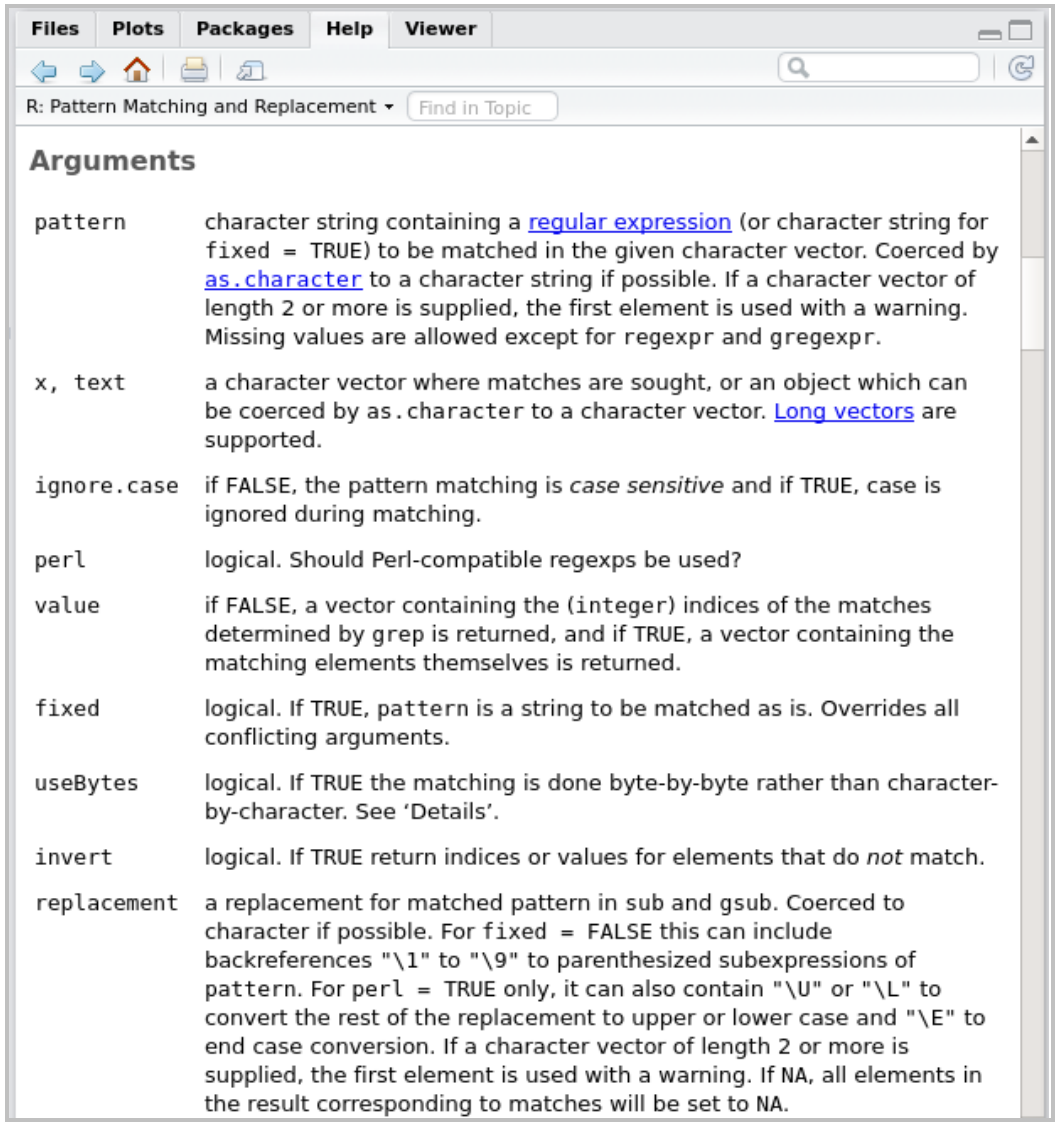
Exercise 1

Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	✔
Use lapply on dataset	
Discuss sapply and use sapply with dataframe	
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Character pattern substitution with `gsub`

```
?gsub

gsub(pattern,      #<- pattern to replace
     replacement,  #<- replacement pattern
     x,            #<- char. string or vector
                   #   where to look for pattern
     ...)
```



Files | Plots | Packages | Help | Viewer

R: Pattern Matching and Replacement — Find in Topic

Arguments

pattern	character string containing a regular expression (or character string for fixed = TRUE) to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed except for regexpr and gregexpr.
x, text	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
ignore.case	if FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching.
perl	logical. Should Perl-compatible regexps be used?
value	if FALSE, a vector containing the (integer) indices of the matches determined by grep is returned, and if TRUE, a vector containing the matching elements themselves is returned.
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See 'Details'.
invert	logical. If TRUE return indices or values for elements that do *not* match.
replacement	a replacement for matched pattern in sub and gsub. Coerced to character if possible. For fixed = FALSE this can include backreferences "\1" to "\9" to parenthesized subexpressions of pattern. For perl = TRUE only, it can also contain "\U" or "\L" to convert the rest of the replacement to upper or lower case and "\E" to end case conversion. If a character vector of length 2 or more is supplied, the first element is used with a warning. If NA, all elements in the result corresponding to matches will be set to NA.

Character pattern substitution with `gsub`

```
# Substitute a character pattern with another one in a character string.
gsub(" ",                                  #<- replace whitespace
     "_",                                  #<- with underscore
     toupper(unique_regions_list[[4]]))  #<- character string, where we want the replacement to occur
```

```
[1] "NORTH_CENTRAL"
```

Making a custom function

What we have

- A character string
- In sentence or lower case
- With white spaces between words

```
Input: "North Central"
```

What we need

- A character string
- In upper case
- With underscores between words

```
Output: "NORTH_CENTRAL"
```

Using custom functions with `lapply`

```
# To put it all together in an `lapply` function, we can do this:
fancy_list = lapply(unique_regions_list,    #<- the original list, which we want to transform
                    function(entry) {       #<- set up function that takes a single argument
                      gsub(" ", "_",        #<- call `gsub` to replace the " " with "_"
                           toupper(entry))  #<- provide the character string that we want transformed
                    })

# Take a look at the output, it did exactly what we wanted!
fancy_list
```

```
[[1]]
[1] "SOUTH"

[[2]]
[1] "WEST"

[[3]]
[1] "NORTHEAST"

[[4]]
[1] "NORTH_CENTRAL"
```

Making a custom function

```
# Copy the function definition from the `lapply`
# function and save it into a variable.
ToUpperAndUnderscore = function(entry) {
  gsub(" ", "_",
       toupper(entry))
}

# Check it out!
ToUpperAndUnderscore(unique_regions_list[[4]])
```

```
[1] "NORTH_CENTRAL"
```

```
# Compare the function output to that produced by the explicit use of `gsub` + `toupper`.
gsub(" ", "_", toupper(unique_regions_list[[4]]))
```

```
[1] "NORTH_CENTRAL"
```

Using custom functions with `lapply`

```
# Now, let's drop our new function into the `lapply` construct
# so that the function is applied to every element of the list.
fancy_list2 = lapply(unique_regions_list, ToUpperAndUnderscore)
fancy_list2
```

```
[[1]]
[1] "SOUTH"

[[2]]
[1] "WEST"

[[3]]
[1] "NORTHEAST"

[[4]]
[1] "NORTH_CENTRAL"
```

- This looks a lot cleaner and modular, which makes your code more readable and re-usable. If you have to use the same function on some other string or a list of strings, you can call it without having to write it all over again!

Generate a random sequence of integers

```
# Set seed to make the random number generator always produce the same result.
set.seed(1)

# Let's create a sample vector.
sample_vec = sample(                #<- generate a sample of numbers from
                    seq(from = 1,   #<- a sequence starting from 1, and
                        to = 50),   #<- ending with 50
                    size = 50)      #<- set the sample size to 50
sample_vec
```

```
 [1] 14 19 28 43 10 41 42 29 27  3  9  7 44 15 48 18 25 33 13 34 47 39 49
[24]  4 30 46  1 40 20  8 31 12 23 38 50 11 36  2 35  5 16  6 26 17 21 22
[47] 24 32 45 37
```

Shuffle data using that sequence

```
state.name
```

```
 [1] "Alabama"        "Alaska"         "Arizona"
"Arkansas"
 [5] "California"     "Colorado"
"Connecticut"    "Delaware"
 [9] "Florida"        "Georgia"        "Hawaii"
"Idaho"
[13] "Illinois"       "Indiana"        "Iowa"
"Kansas"
[17] "Kentucky"       "Louisiana"      "Maine"
"Maryland"
[21] "Massachusetts"  "Michigan"
"Minnesota"      "Mississippi"
[25] "Missouri"       "Montana"
"Nebraska"       "Nevada"
[29] "New Hampshire"  "New Jersey"     "New
Mexico"     "New York"
[33] "North Carolina" "North Dakota"   "Ohio"
"Oklahoma"
[37] "Oregon"         "Pennsylvania"   "Rhode
Island"   "South Carolina"
[41] "South Dakota"   "Tennessee"      "Texas"
"Utah"
[45] "Vermont"        "Virginia"
"Washington"     "West Virginia"
[49] "Wisconsin"      "Wyoming"
```

```
state.name[sample_vec]
```

```
 [1] "Indiana"        "Maine"          "Nevada"
"Texas"
 [5] "Georgia"        "South Dakota"
"Tennessee"      "New Hampshire"
 [9] "Nebraska"       "Arizona"        "Florida"
"Connecticut"
[13] "Utah"           "Iowa"           "West
Virginia"  "Louisiana"
[17] "Missouri"       "North Carolina"
"Illinois"       "North Dakota"
[21] "Washington"     "Rhode Island"
"Wisconsin"      "Arkansas"
[25] "New Jersey"     "Virginia"       "Alabama"
"South Carolina"
[29] "Maryland"       "Delaware"       "New
Mexico"     "Idaho"
[33] "Minnesota"      "Pennsylvania"   "Wyoming"
"Hawaii"
[37] "Oklahoma"       "Alaska"         "Ohio"
"California"
[41] "Kansas"         "Colorado"       "Montana"
"Kentucky"
[45] "Massachusetts"  "Michigan"
"Mississippi"    "New York"
[49] "Vermont"        "Oregon"
```

Separate even observations from odd

- Let's take this a step further and say we need to separate generated indices into odd and even. We could check each entry within a vector for divisibility by two and return `TRUE` or `FALSE`

```
# The first number in `sample_vec` is 14.
sample_vec[1]
```

```
[1] 14
```

```
# Check if it is even.
if(sample_vec[1] %% 2 == 0){  #<- number `%%` 2 will produce either `0` or `1`
  TRUE                        #<- return TRUE if the remainder is `0`
} else{                        #<- otherwise
  FALSE                       #<- return `0`
}
```

```
[1] TRUE
```

Transforming a vector of numbers with `lapply`

- Now we can generalize the above code into a function that would check if the number is even first, and then use `lapply` to apply that function to each entry in the vector

```
IsEven = function(number){   #<- the function takes a number as input
  if(number %% 2 == 0){      #<- checks if the remainder after division by 2 is zero
    TRUE                     #<- returns `TRUE` if it is
  }else{                     #<- otherwise
    FALSE                    #<- it returns `FALSE`
  }
}

# Use the function within the `lapply` construct to check if each
# number in the vector is even.
logical_list = lapply(sample_vec, IsEven)
head(logical_list, 4) #<- Take a look at the first 4 elements of the output list.
```

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

[[2]]
[1] FALSE

[[3]]
[1] TRUE

[[4]]
[1] FALSE
```

Flattening output lists into vectors

- We can easily flatten the output list produced by `lapply` by wrapping it into a `unlist` function, which flattens the list into a vector

```
# Flatten the list into a vector.
logical_vec = unlist(logical_list)

# Check the result!
str(logical_vec)
```

```
 logi [1:50] TRUE FALSE TRUE FALSE TRUE FALSE ...
```

Flattening output lists into vectors

```
# Keep only even indices.
even_ids = sample_vec[logical_vec]
even_ids
```

```
 [1] 14 28 10 42 44 48 18 34  4 30 46 40 20  8 12 38 50 36  2 16  6 26 22
[24] 24 32
```

```
# Let's subset the vector of state names
# using the resulting vector of even randomly
# distributed numbers.
state.name[even_ids]
```

```
 [1] "Indiana"        "Nevada"         "Georgia"        "Tennessee"
 [5] "Utah"           "West Virginia"  "Louisiana"      "North Dakota"
 [9] "Arkansas"       "New Jersey"     "Virginia"       "South Carolina"
[13] "Maryland"       "Delaware"       "Idaho"          "Pennsylvania"
[17] "Wyoming"        "Oklahoma"       "Alaska"         "Kansas"
[21] "Colorado"       "Montana"        "Michigan"       "Mississippi"
[25] "New York"
```

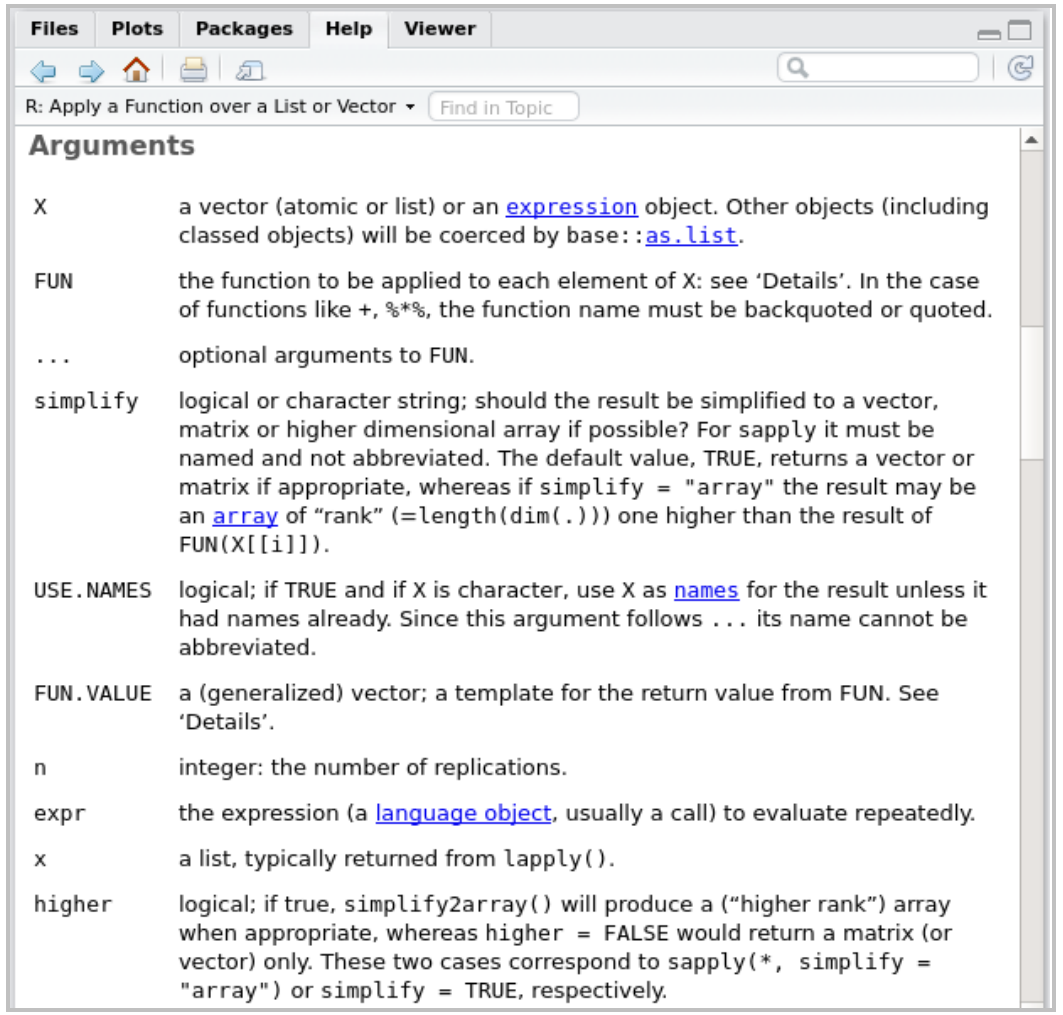
Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	✔
Use lapply on dataset	✔
Discuss sapply and use sapply with dataframe	
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Variants of `lapply`

- One of the variants of `lapply` is `sapply`, which takes the same main arguments as `lapply` and returns a **vector** instead of a **list**

```
?sapply

sapply(X,          #<- Input a list or a vector
       FUN, ...)  #<- and function
```



Simplified `lapply` = `sapply`

```
# Apply `IsEven` function to a vector
# using `lapply`.
logical_list = lapply(sample_vec, IsEven)

# Flatten the list into a vector.
logical_vec = unlist(logical_list)

# Check the result!
str(logical_vec)
```

```
 logi [1:50] TRUE FALSE TRUE FALSE TRUE FALSE
...
```

```
# Apply `isEven` function to a vector
# using `sapply`.
logical_vec2 = sapply(sample_vec, IsEven)



# Check the result!
str(logical_vec2)
```

```
 logi [1:50] TRUE FALSE TRUE FALSE TRUE FALSE
...
```

Using `sapply` with dataframes

The `apply` family of functions not only works with lists and vectors, but also allows us to perform operations on dataframes and matrices. Let's apply a set of transformations to the `state_data` matrix before we demonstrate `sapply`'s capabilities.

1. Convert `state_data` to a dataframe
2. Add `State` column from the `state_data`'s row names
3. Reset all current row names in the dataframe
4. Check class of each variable in the dataset

Transforming `state_data`

What we have

- A matrix with row names and without `State` variable

state.x77 ×

Filter

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945
California	21198	5114	1.1	71.71	10.3	62.6	20	156361
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766
Connecticut	3100	5348	1.1	72.48	3.1	56.0	139	4862
Delaware	579	4809	0.9	70.06	6.2	54.6	103	1982
Florida	8277	4815	1.3	70.66	10.7	52.6	11	54090
Georgia	4931	4091	2.0	68.54	13.9	40.6	60	58073
Hawaii	868	4963	1.9	73.60	6.2	61.9	0	6425
Idaho	813	4119	0.6	71.87	5.3	59.5	126	82677
Illinois	11197	5107	0.9	70.14	10.3	52.6	127	55748

Showing 1 to 14 of 50 entries

What we need

- A dataframe without row names and with `State` variable

state_df ×

Filter

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
1	3615	3624	2.1	69.05	15.1	41.3	20	50708	Alabama
2	365	6315	1.5	69.31	11.3	66.7	152	566432	Alaska
3	2212	4530	1.8	70.55	7.8	58.1	15	113417	Arizona
4	2110	3378	1.9	70.66	10.1	39.9	65	51945	Arkansas
5	21198	5114	1.1	71.71	10.3	62.6	20	156361	California
6	2541	4884	0.7	72.06	6.8	63.9	166	103766	Colorado
7	3100	5348	1.1	72.48	3.1	56.0	139	4862	Connecticut
8	579	4809	0.9	70.06	6.2	54.6	103	1982	Delaware
9	8277	4815	1.3	70.66	10.7	52.6	11	54090	Florida
10	4931	4091	2.0	68.54	13.9	40.6	60	58073	Georgia
11	868	4963	1.9	73.60	6.2	61.9	0	6425	Hawaii
12	813	4119	0.6	71.87	5.3	59.5	126	82677	Idaho
13	11197	5107	0.9	70.14	10.3	52.6	127	55748	Illinois

Showing 1 to 13 of 50 entries

Converting a matrix into a dataframe

```
# 1. We need to convert our matrix to a dataframe.
state_df = as.data.frame(state_data)
class(state_df)
```

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

```
str(state_df)
```

```
'data.frame':   50 obs. of  8 variables:
 $ Population: num  3615 365 2212 2110 21198 ...
 $ Income    : num  3624 6315 4530 3378 5114 ...
 $ Illiteracy: num  2.1 1.5 1.8 1.9 1.1 0.7 1.1 0.9 1.3 2 ...
 $ Life Exp  : num  69 69.3 70.5 70.7 71.7 ...
 $ Murder    : num  15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 10.7 13.9 ...
 $ HS Grad   : num  41.3 66.7 58.1 39.9 62.6 63.9 56 54.6 52.6 40.6 ...
 $ Frost     : num  20 152 15 65 20 166 139 103 11 60 ...
 $ Area      : num  50708 566432 113417 51945 156361 ...
```

Adding a column to a dataframe

- We need to create a variable `State` (i.e. the name of the variable in the two datasets has to be the same, since this is the variable that we will use to merge this dataframe by)

```
# 2. The names of the rows of the dataframe will now become a column `State`.
state_df$State = rownames(state_df)  #<- to add a named column to a dataframe, we can use the
                                     #   `$` operator followed by the new column name
str(state_df)
```

```
'data.frame':   50 obs. of  9 variables:
 $ Population: num  3615 365 2212 2110 21198 ...
 $ Income    : num  3624 6315 4530 3378 5114 ...
 $ Illiteracy: num  2.1 1.5 1.8 1.9 1.1 0.7 1.1 0.9 1.3 2 ...
 $ Life Exp  : num  69 69.3 70.5 70.7 71.7 ...
 $ Murder    : num  15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 10.7 13.9 ...
 $ HS Grad   : num  41.3 66.7 58.1 39.9 62.6 63.9 56 54.6 52.6 40.6 ...
 $ Frost     : num  20 152 15 65 20 166 139 103 11 60 ...
 $ Area      : num  50708 566432 113417 51945 156361 ...
 $ State     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
```

Reset row names of a dataframe to indices

```
# Take a look at the
# current dataframe that
# contains the row names.
View(state_df)
```

state_df

Filter

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708	Alabama
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432	Alaska
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417	Arizona
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945	Arkansas
California	21198	5114	1.1	71.71	10.3	62.6	20	156361	California
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766	Colorado
Connecticut	3100	5348	1.1	72.48	3.1	56.0	139	4862	Connecticut
Delaware	579	4809	0.9	70.06	6.2	54.6	103	1982	Delaware
Florida	8277	4815	1.3	70.66	10.7	52.6	11	54090	Florida
Georgia	4931	4091	2.0	68.54	13.9	40.6	60	58073	Georgia
Hawaii	868	4963	1.9	73.60	6.2	61.9	0	6425	Hawaii
Idaho	813	4119	0.6	71.87	5.3	59.5	126	82677	Idaho
Illinois	11197	5107	0.9	70.14	10.3	52.6	127	55748	Illinois

Showing 1 to 13 of 50 entries

```
# Reset row names to their default values
# (i.e. indices of rows).
rownames(state_df) = NULL
```

```
View(state_df)
```

state_df

Filter

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
1	3615	3624	2.1	69.05	15.1	41.3	20	50708	Alabama
2	365	6315	1.5	69.31	11.3	66.7	152	566432	Alaska
3	2212	4530	1.8	70.55	7.8	58.1	15	113417	Arizona
4	2110	3378	1.9	70.66	10.1	39.9	65	51945	Arkansas
5	21198	5114	1.1	71.71	10.3	62.6	20	156361	California
6	2541	4884	0.7	72.06	6.8	63.9	166	103766	Colorado
7	3100	5348	1.1	72.48	3.1	56.0	139	4862	Connecticut
8	579	4809	0.9	70.06	6.2	54.6	103	1982	Delaware
9	8277	4815	1.3	70.66	10.7	52.6	11	54090	Florida
10	4931	4091	2.0	68.54	13.9	40.6	60	58073	Georgia
11	868	4963	1.9	73.60	6.2	61.9	0	6425	Hawaii
12	813	4119	0.6	71.87	5.3	59.5	126	82677	Idaho
13	11197	5107	0.9	70.14	10.3	52.6	127	55748	Illinois

Showing 1 to 13 of 50 entries

Using `sapply` with dataframes

```
# Look up the class of each variable in the state data and save into a vector.
variable_class = sapply(state_df, class)
variable_class
```

```
 Population      Income  Illiteracy    Life Exp      Murder     HS Grad
  "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
      Frost        Area       State
  "numeric"   "numeric" "character"
```

Apply or not apply

When to use functionals ✔

- When there is a `for` loop that performs the same operation on every element of a vector, list or matrix
- When each operation on individual elements of the array is independent of each other
- When the result of the modification after the operation is performed is stored separately from the original data structure

When NOT to use functionals ✘

- When there is a `while` loop that stops when a certain condition has been met
- When each successive operation on the element is dependent on a previous one (such behavior is often related to *recursion*)
- When the result of the operations performed on the elements of a data structure is stored within the data structure itself or the operation is modifying the existing data structure in place

Knowledge check 2

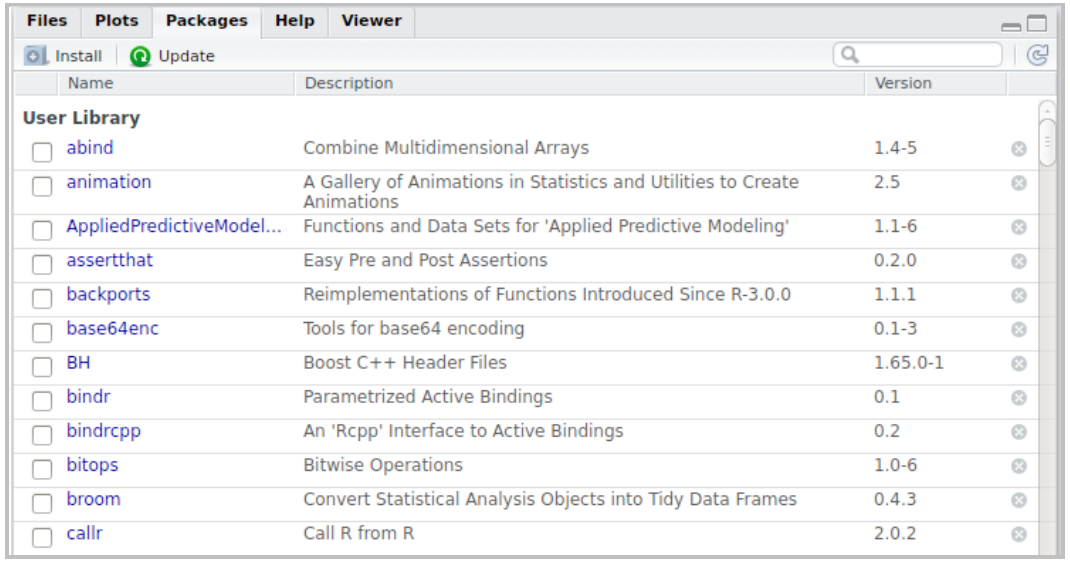
Exercise 2

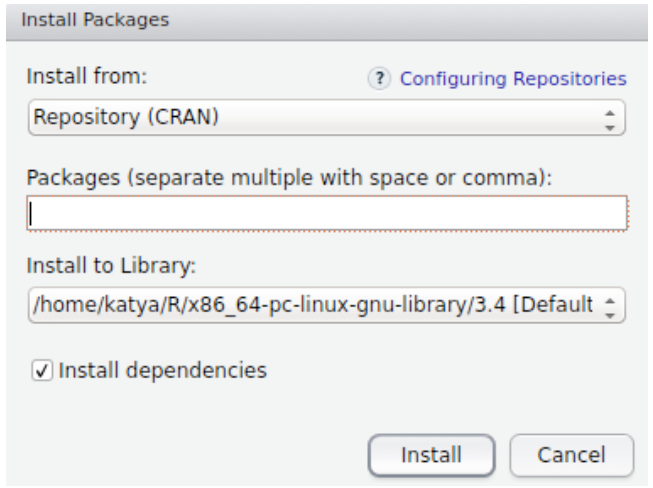
Module completion checklist

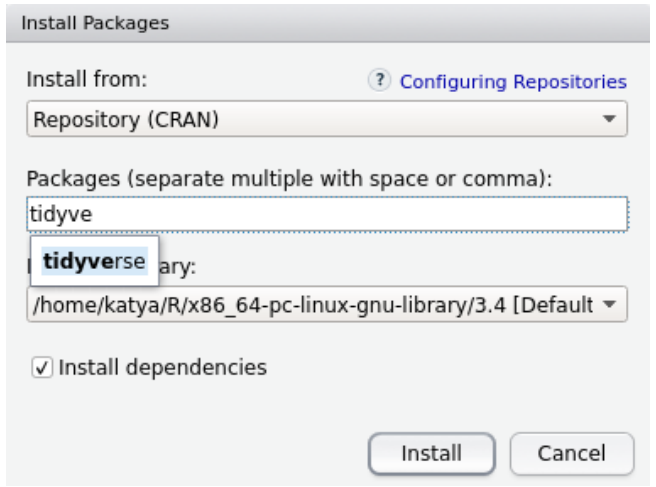
Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	✔
Use lapply on dataset	✔
Discuss sapply and use sapply with dataframe	✔
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Installing packages with package explorer

- RStudio has a built-in package manager in its bottom right pane
- Click on `Packages` tab in the bottom-right pane
- Click `Install`
- Type package name and install

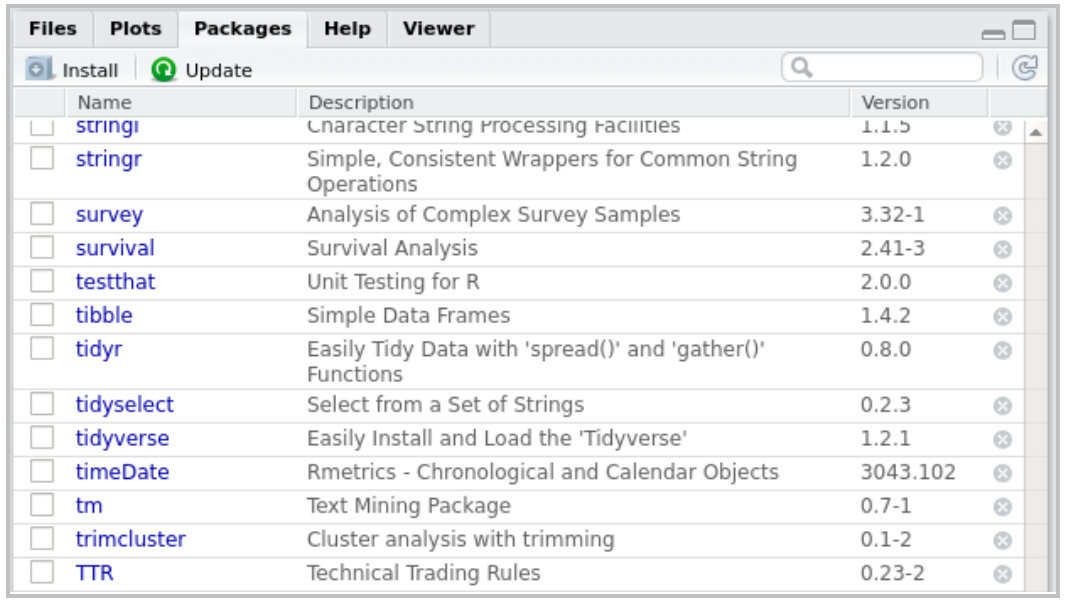




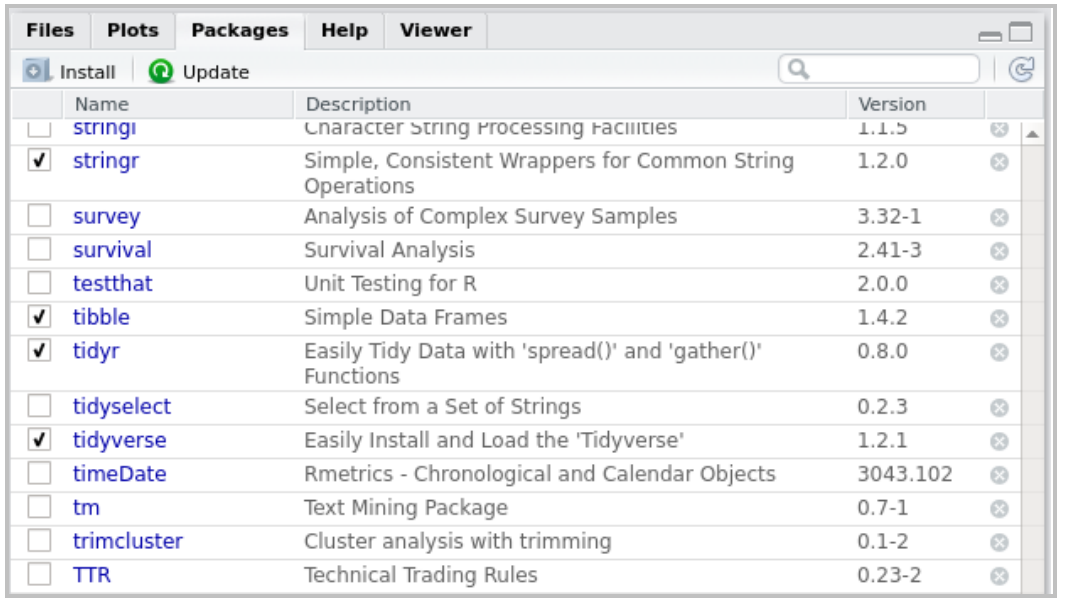


Load package

- The installed package should appear in the list of packages in the explorer



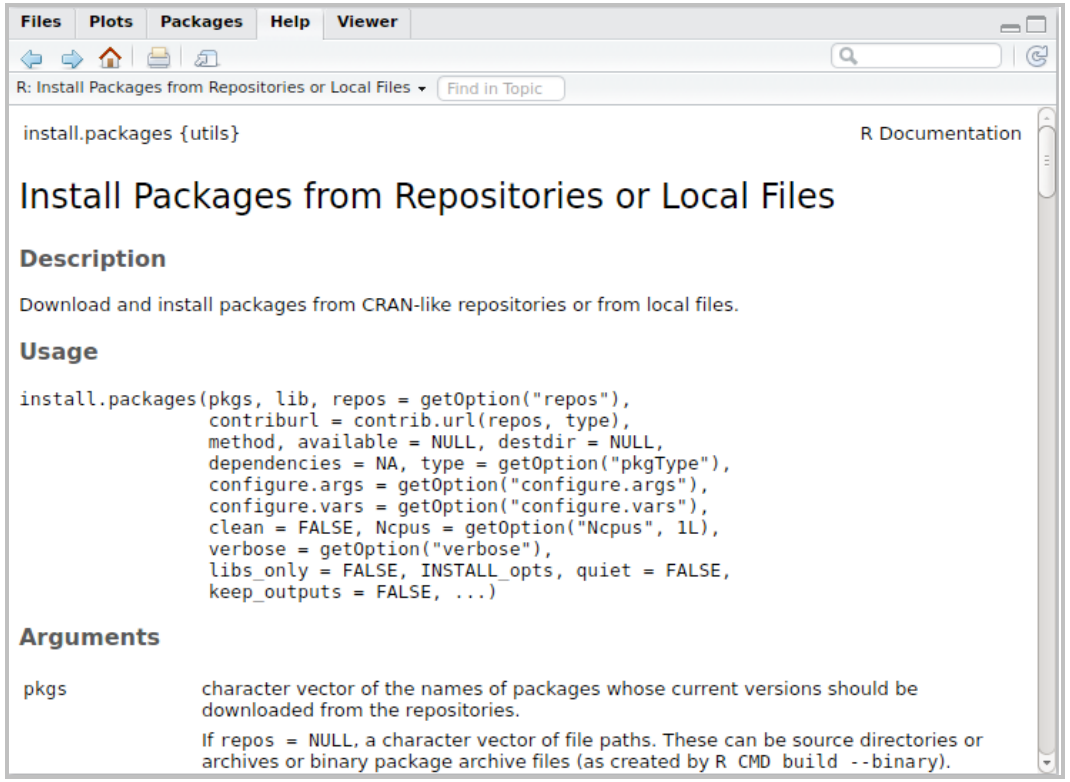
- To load the package into R's environment, click the box next to the desired package



Installing packages

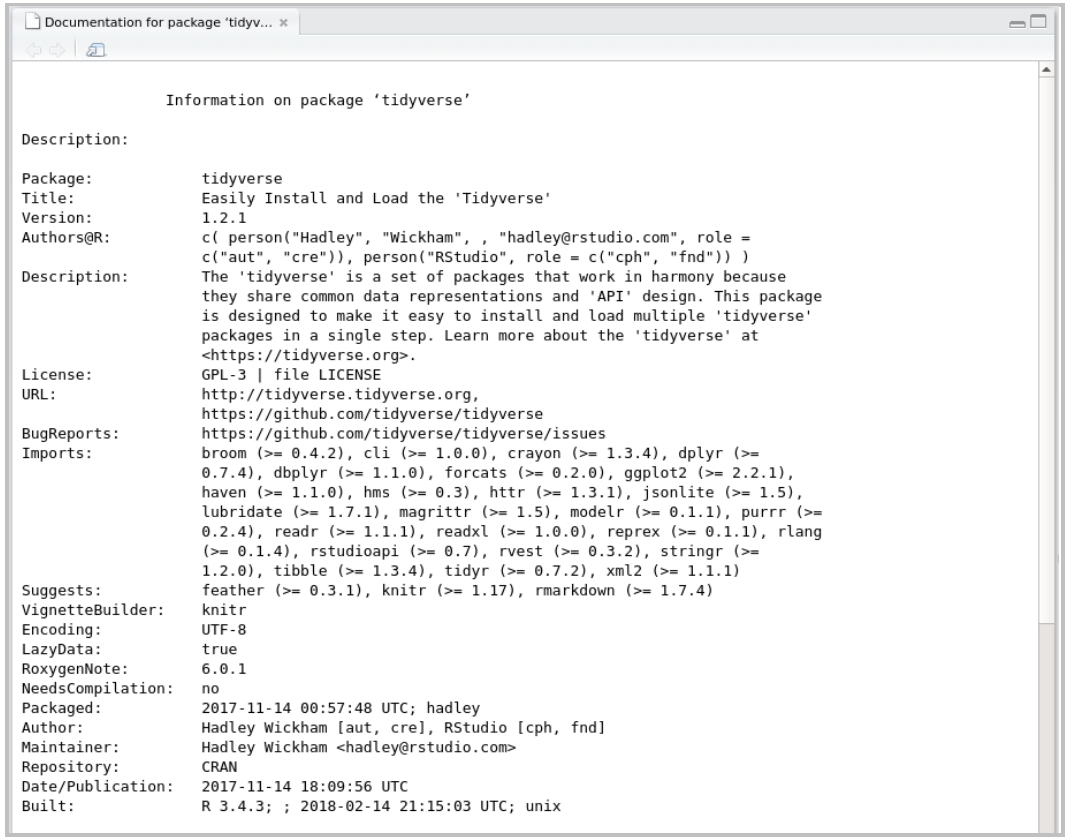
```
# Install package.
?install.packages
```

- If we want to use a function that comes from a `package`, we need to install the `package`
- We need to provide a single required argument: a character string corresponding to the `package name`



Installing packages

```
# Install package.
install.packages("tidyverse")

# Load the package into the environment.
library(tidyverse)

# View package documentation.
library(help = "tidyverse")
```



Installing packages and loading data

- To review the functions within tidyverse, we will be working with multiple datasets found through libraries
- Let's install and load the `nycflights` package
- We will use this dataset to illustrate `dplyr` functions

```
# install.packages("nycflights13")
library(nycflights13)
```

Defining packages and libraries

- **Packages** are *collections* of R functions, data, and compiled code

- **Libraries** are the *directories* where packages are stored

Typically, you use the library() function to load a package in your current session

Directory settings

- Let's make sure our directories are set correctly, so we don't have to worry about this throughout the course

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac/Linux).
main_dir = "~/Desktop/af-werx"

# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/af-werx"

# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = paste0(main_dir, "/data")

# Make `plots_dir` from the `main_dir` and
# remainder of the path to plots directory.
plot_dir = paste0(main_dir, "/plots")
```

Set working directory and load data

```
setwd(data_dir)
load("tidyr_tables.RData")
```

Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	✔
Use lapply on dataset	✔
Discuss sapply and use sapply with dataframe	✔
Demonstrate installing a package and loading a library	✔
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	

Introduction to data transformation with tidyverse

- When you are given 'messy' data, your goal is to transform it to a usable format
- To do this, you need multiple packages that are found within the universe of `tidyverse`
- We are going to go over how to:
  - manipulate data with : `dplyr`
  - transform data with: `tidyr`

Packages in the tidyverse change fairly frequently. You can see if updates are available and optionally install them by running `tidyverse_update`.

Data transformation with dplyr

- `dplyr` is an essential package within the tidyverse universe
- It will be the tool we use for transforming your data by filtering, aggregating and summarizing your data
- Before starting this lesson, we want to make sure to let you know that dplyr does overwrite some `base r` packages: `filter` and `lag`
- If you have loaded dplyr and want to use the base version of the package, you will have to type in the full name: `stats::filter` and `stats::lag`

Load the dataset

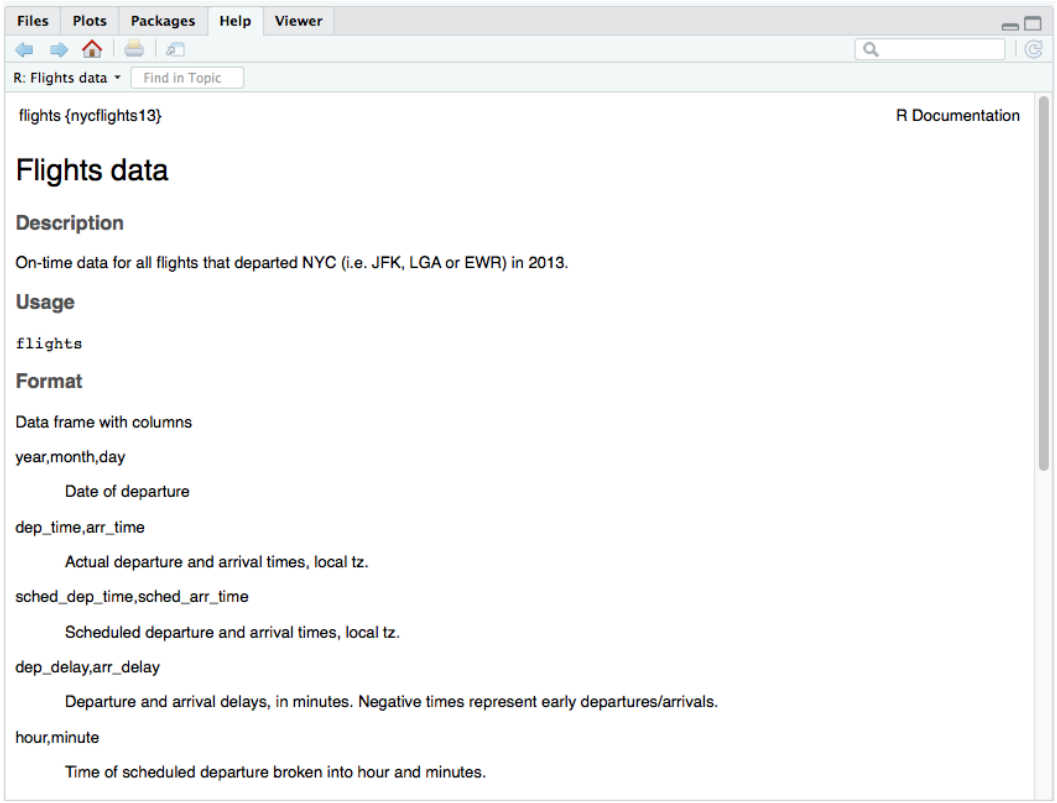
- Let's look at the dataset we will be working with - `flights`

```
# Load the dataset and save it as 'flights'.
# It is native to R so we can load it like this.
flights = nycflights13::flights
```

Get dataset information

- You can find documentation for the dataset like this:

```
?flights
```



Basics of dplyr

- There are six functions that provide the verbs for the language of data manipulation
- They will most definitely make your life as a data scientist easier
- They are:

Function	Use Case	Data Type
`filter`	Pick observations by their value	All data types
`arrange`	Reorder the rows	All data types
`select`	Pick variables by their names	All data types
`mutate`	Create new variables with functions of existing variables	All data types
`summarise`	Collapse many values down to a single summary	All data types
`group_by`	Allows the first five functions to operate on a dataset group by group	All data types

Framework of dplyr

This is the framework of `dplyr`:

1. The first argument is a dataframe
2. The next arguments describe what to do with the dataframe, using the six key `dplyr` functions
3. The final result is a new, transformed dataframe!

We will go into more detail about how each of these six verbs work!

Knowledge check 3

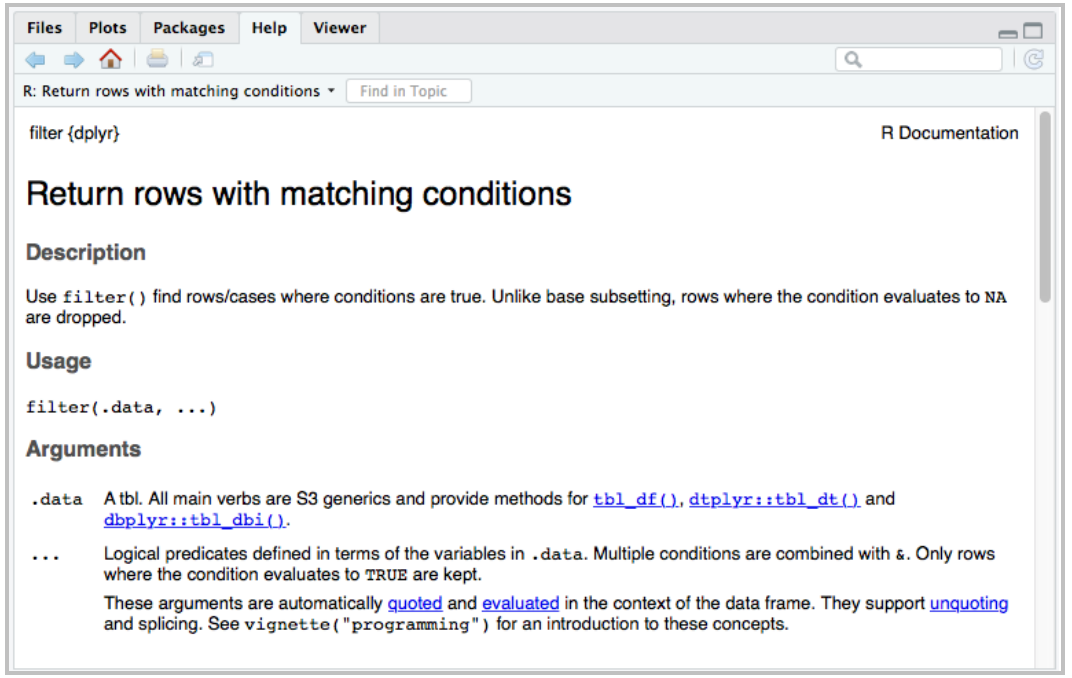
Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	✔
Use lapply on dataset	✔
Discuss sapply and use sapply with dataframe	✔
Demonstrate installing a package and loading a library	✔
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	✔
Apply the filter function to subset data	

Filtering flights

- `filter` allows you to subset observations based on their values

```
?dplyr::filter

filter(df,            #<- dataframe
       filter_cond1,  #<- subsetting rule(s)
       ...)           #<- other arguments
```



Save your subset

- Let's say that you want to see all flights from January 2013

```
# Load the flights dataset into the environment.
data(flights)

# Filter `flights` dataframe to display all records
# from January (month == 1) of 2013 (year == 2013).
filter(flights,       #<- set data
       month == 1,    #<- filter by month
       year == 2013) #<- filter by year
```

```
# A tibble: 27,004 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# … with 26,994 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

More general operators

- If you want to build on top of the filtered dataset, you will need to save your new subset

```
# You will have to make sure to save the subset. To do this, use `=`.
filter_flights = filter(flights, month == 1, day == 25)

# View your output.
filter_flights
```

```
# A tibble: 922 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1    25       15           1815       360      208
 2  2013     1    25       17           2249        88      119
 3  2013     1    25       26           1850       336      225
 4  2013     1    25      123           2000       323      229
 5  2013     1    25      123           2029       294      215
 6  2013     1    25      456            500        -4      632
 7  2013     1    25      519            525        -6      804
 8  2013     1    25      527            530        -3      820
 9  2013     1    25      535            540        -5      826
10  2013     1    25      539            540        -1     1006
# … with 912 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

Filter options

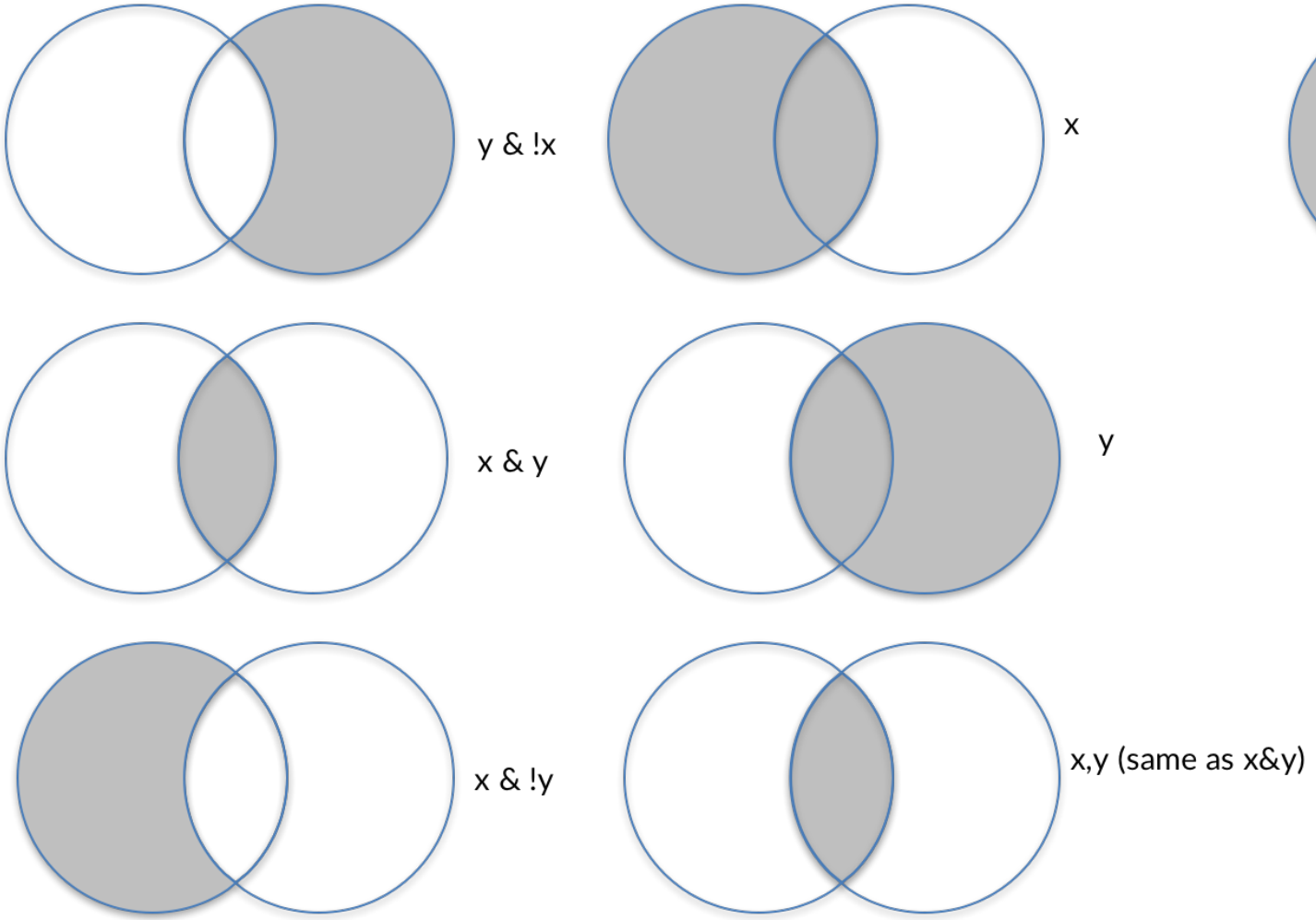
- You can use the standard filtering operations when working with integer data types

Operation	Use Case	Example
>	Greater than	6 > 4
>=	Greater than or equal to	4 >= 4
<	Less than	4 < 6
<=	Less than or equal to	4 <= 4
!=	Not equal to	4 != 6
==	Equal to	4 == 4

- And more general operators:

Operation	Use Case	Example
OR or |	either can be true to satisfy	x == 4 OR x == 12, x == 2 | x == 13
and, &	and, both need to be true	x == 4 & y == 2
!	Not true, inverse selection	x != 4
%in%	value in the following list of values	x %in% c(4, 16, 32)

Examples of logical operators



Examples of logical operators

- What if we want to see all flights from January **and** on the 25th?

Note: after running each example, we will record the number of rows. This will help illustrate each operator and how a simple change of one Boolean operator can make a big difference.

```
# Filter with just `&`.
filter(flights, month == 1 & day == 25)
```

```
# A tibble: 922 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1    25       15           1815       360      208
 2  2013     1    25       17           2249        88      119
 3  2013     1    25       26           1850       336      225
 4  2013     1    25      123           2000       323      229
 5  2013     1    25      123           2029       294      215
 6  2013     1    25      456            500        -4      632
 7  2013     1    25      519            525        -6      804
 8  2013     1    25      527            530        -3      820
 9  2013     1    25      535            540        -5      826
10  2013     1    25      539            540        -1     1006
# … with 912 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

Examples of logical operators continued

- What if we want to see all flights, but *exclude* those from January and those on the 25th?

```
# Filter with `!`.
filter(flights, month != 1 & day != 25)
```

```
# A tibble: 299,597 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013    10     1      447            500       -13      614
 2  2013    10     1      522            517         5      735
 3  2013    10     1      536            545        -9      809
 4  2013    10     1      539            545        -6      801
 5  2013    10     1      539            545        -6      917
 6  2013    10     1      544            550        -6      912
 7  2013    10     1      549            600       -11      653
 8  2013    10     1      550            600       -10      648
 9  2013    10     1      550            600       -10      649
10  2013    10     1      551            600        -9      727
# … with 299,587 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

- Here, we are looking for all flights that are **not in January** and **not on the 25th**; total number of rows should be **299,597**

Examples of logical operators continued

```
# Filter with `%in%`.
filter(flights, month %in% c(1, 2) & day == 25)
```

```
# A tibble: 1,883 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1    25       15           1815       360      208
 2  2013     1    25       17           2249        88      119
 3  2013     1    25       26           1850       336      225
 4  2013     1    25      123           2000       323      229
 5  2013     1    25      123           2029       294      215
 6  2013     1    25      456            500        -4      632
 7  2013     1    25      519            525        -6      804
 8  2013     1    25      527            530        -3      820
 9  2013     1    25      535            540        -5      826
10  2013     1    25      539            540        -1     1006
# … with 1,873 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

- This is a combination of `&` and `%in%` subsetting **all flights from January and February** that are **on the 25th**; number of rows should be **1,883**

Using filter with NA values

- `filter` only includes rows where the condition is TRUE; it excludes both FALSE and NA values
- If you want to preserve missing values, ask for them explicitly:

```
# Create a dataframe with 2 columns.
NA_df = data.frame(x = c(1, NA, 2),  #<- column x with 3 entries with 1 NA
                   y = c(1, 2, 3))   #<- column y with 3 entries

# Filter without specifying anything regarding NAs.
filter(NA_df, x >= 1)
```

```
  x y
1 1 1
2 2 3
```

```
# Filter with specifying to keep rows if there is an NA.
filter(NA_df, is.na(x) | x >= 1)
```

```
   x y
1  1 1
2 NA 2
3  2 3
```

Knowledge check 4

Exercise 3

Module completion checklist

Objective	Complete
Discuss and examine state.x77 dataset	✔
Demonstrate working with the random number generator	✔
Explain apply family of functions as an alternative to for loops	✔
Use lapply on dataset	✔
Discuss sapply and use sapply with dataframe	✔
Demonstrate installing a package and loading a library	✔
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	✔
Apply the filter function to subset data	✔

Workshop!

- **Today will be your first *after class* workshop**
- Workshops are to be completed outside of class and emailed to the instructor by the beginning of class tomorrow
- Make sure to comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Workshop objectives:
  - Use apply family of functions to manipulate data
  - Generate random rows/columns from the dataset of your choice by using random number generator
  - Examine the dataset by using `filter` function and analyze the data

This completes our module

Congratulations!