Fundamentals of R

Adam Kuczynski

UNIVERSITY OF WASHINGTON

Objects

To create an object, you can use:

- `<-` `<<-` assignment (right to left)
- `->` `->>` rightwards assignment
- `=` assignment (right to left)

99.9% of the time you will use `<-`:

```
object <- value
```

You should *never* use `=` for object assignment. The reason is that `=` calls two different functions depending on the context:

```
mean(x = c(1, 2, 3, 4, 5))
```

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

```
print(x)
```

```
## Error in print(x): object 'x'
```

```r
mean(y <- c(1, 2, 3, 4, 5))
```

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

```r
print(y)
```

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

You can fix this by telling R that your `=` is meant to be an assignment operator, not argument assignment:

```r
mean((z = c(1, 2, 3, 4, 5)))
```

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

```r
print(z)
```

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

You can also assign object left-to-right using `->`, but this is very unusual and only done in unique circumstances where doing so improves readability of our code:

```r
value -> object
```

We'll get back to `<<-` and `->>` later in the course when we learn how to create our own functions. For now you can forget about them!

Comments

Comments are a very important part of writing R code! When R sees a comment, it will skip over trying to interpret (i.e., run) that code.

Use the `#` to write a comment in your R script:

```
# This code is a comment, so R doesn't run it as code
```

```
This code is not a comment, so R tries to run it as code
```

```
## Error: <text>:1:6: unexpected symbol
## 1: This code
##           ^
```

Commenting your code helps you:

- Remember what a chunk of code does when you return to the code
- Communicate with collaborators
- Keep your code organized

Atomic Types

R, like other programming languages, keeps track of the type of object you are working with. It does this much in the same way that we do in our own lives. For example, we know that our age is a number and our name is a bunch of characters squished together.

There are *four* main atomic types in R (there are more, but you most likely won't use them)

To check the type of a value in R, you can use the `typeof()` function. For example:

```
typeof("Psych548")
```

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

```
typeof(TRUE)
```

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

1. Numeric

Numeric vectors (more on these later) are numbers, either *integers* or *doubles* (i.e., floating point numbers, decimals)

- Integers: 1, 2, 3, 4, etc.
- Doubles: 1.0, 2.43, 3.92, 4.0934853409

To check directly if something is numeric, an integer, or a double:

- `is.numeric(x)`
- `is.integer(x)`
- `is.double(x)`

You can also coerce (i.e., tell R) that you want to store a value as a numeric object:

- `as.numeric(x)`
- `as.integer(x)`
- `as.double(x)`

Note: Unless you say otherwise, R will store all numeric values as doubles:

```
typeof(10)
```

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

You can specify a single integer using a whole number following by `L`:

```
is.integer(10)
```

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

```
is.integer(10L)
```

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

All integers and doubles are numeric:

```
is.numeric(30)
```

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

```
is.numeric(30L)
```

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

2. Character

Character vectors contain **strings** of characters squished together using quotations. For example:

```
"This is a string"
```

```
## [1] "This is a string"
```

```
"هذه سلسلة "
```

```
## [1] "هذه سلسلة "
```

```
"นี่คือสตริง"
```

```
## [1] "นี่คือสตริง"
```

This is also, technically, a string (character type):

```
""
```

To check directly if something is of type character:

- `is.character(x)`

You can also coerce values into characters using:

- `as.character(x)`

For example:

```
as.character(1234)
```

```
## [1] "1234"
```

To check how many characters are in a string, you can use the `nchar()` function:

```
nchar("Psych548")
```

```
## [1] 8
```

To squish two strings together, use the `paste()` function:

```
paste("Adam", "Kuczynski")
```

```
## [1] "Adam Kuczynski"
```

To insert formatted values into strings, use `sprintf()`[1]:

```
weeknum <- 2
weekchar <- "two"
sprintf("This is Week %d (%s) of the quarter.", weeknum, weekchar)
```

```
## [1] "This is Week 2 (two) of the quarter."
```

`glue()` will make your life a lot easier!

```
library(glue)
glue("This is Week {weeknum} ({weekchar}) of the quarter.")
```

```
## This is Week 2 (two) of the quarter.
```

[1] see `?sprintf` for more information. This is a really powerful function that lets you do things like format values dynamically.

3. Logical

Logical values are `TRUE`, `FALSE`, and `NA` (more on this later).

Logical types *must* be capitalized. `True` is not the same as `TRUE`.

Logical types are also commonly represented as one uppercase letter:

- `TRUE`, `T`
- `FALSE`, `F`

To check directly if something is logical type:

- `is.logical(x)`

You can also coerce values into logical types:

- `as.logical(x)`

```
is.logical(TRUE)
```

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

Underneath the hood, R stores logical values as 0s (`FALSE`) and 1s (`TRUE`). This means you can do math with logical values!

```
as.numeric(c(TRUE, FALSE))
```

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

```
TRUE + TRUE
```

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

```
FALSE + FALSE + TRUE
```

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

```
50 / TRUE
```

```
## [1] 50
```

```
TRUE*3 / 2 + 50^F
```

```
## [1] 2.5
```

4. Factor

Factors are a special type of variable that is used to denote categorical values. These are *extremely* useful when analyzing categorical data, because R will do all the dummy coding for you.

Underneath the hood, factors are stored as numeric values with a table of corresponding levels. This means that factors take up less memory than characters, and comparing factors will be faster than comparing strings (because R only needs to compare numbers).[1]

To check directly if something is a factor:

- `is.factor(x)`

To coerce something into a factor:

- `as.factor(x)`

[1] This likely will not make a difference for you though! The most important consideration in determining whether or not an object should be a factor is whether it is categorical.

Coercing a character vector into a factor is easy because R already knows the labels for each level:

```
areas <- c("Clinical", "Social", "Cog/Per", "Developmental", "BNS", "Animal Behavi
areas <- as.factor(areas)
print(areas)
```

```
## [1] Clinical        Social          Cog/Per         Developmental
## [5] BNS             Animal Behavior
## Levels: Animal Behavior BNS Clinical Cog/Per Developmental Social
```

--

However, often you will have a numeric vector that represents categorical data. To change numeric vectors into factors:

```
agegroup <- c(0, 1, 0, 2, 1, 0)
agegroup <- factor(x = agegroup,
                   levels = c(0, 1, 2),
                   labels = c("Child", "Adolescent", "Adult"))
print(agegroup)
```

```
## [1] Child      Adolescent Child      Adult      Adolescent Child
## Levels: Child Adolescent Adult
```

Logical and Relational Operators

What are operators?

Logical operators are foundational to programming in R and allow you to compare two values together to control your programming logic.

Logical operators *always* return a logical value (`TRUE`, `FALSE`, or `NA`), and are most commonly used to subset data (more on this later) and control the flow if your code `if`/`else` statements (more on this later as well!)

Relational Operators:

- `<`, `>`,
- `>=`, `<=`,
- `==`, `!=`

Logical operators:

- `&`, `&&`
- `|`, `||`
- `!`

Relational Operators

`>` and `<` return `TRUE` if the left side is greater than (`>`) or less than (`<`) the right side, otherwise they return `FALSE`

```
200 > 300
```

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

```
300 > 200
```

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

```
200 < 300
```

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

```
300 < 200
```

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

`>=` and `<=` return `TRUE` if the left side is greater than or equal to (`>=`) or less than or equal to (`<=`) the right side, otherwise they return `FALSE`

```
300 > 200
```

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

```
300 >= 300
```

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

```
300 <= 200
```

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

```
200 <= 300
```

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

`==` and `!=` return `TRUE` if the left side is equal to (`==`) or not equal to (`!=`) the right side, otherwise they return `FALSE`

```
200 == 200
```

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

```
200 == 300
```

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

```
200 != 200
```

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

```
200 != 300
```

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

Comparing strings

You can also use relational operators to compare strings:

```
"This" == "This"
```

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

```
"This" == "That"
```

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

```
"This" != "That"
```

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

Be careful though! These comparisons are case sensitive:

```
"this" == "THIS"
```

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

Things get weird though when you use other relational operators with strings

Shouldn't this throw an error?

```
"UW" > "WSU"
```

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

R *can* compare strings in this way,
but this doesn't mean you should!

What does this even mean?!

```
"A" > 10
```

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

```
11 > "10"
```

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

and, and and

The `&` and `&&` ('and', 'and and') operators are used to compare whether two (or more) conditions are `TRUE`

If both conditions are `TRUE`, `TRUE` is returned, otherwise `FALSE` (or `NA`[1]) is returned


```
TRUE && TRUE
```

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

```
TRUE && FALSE
```

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

[1] Caution: If one of the terms is `NA`, `&&` will return `NA`.

You can combine multiple `&&` to check that *all* conditions evaluate to `TRUE`

```
TRUE && TRUE && T && TRUE
```

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

```
TRUE && FALSE && TRUE && FALSE
```

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

`&&` is more helpful when we can actually evaluate certain conditions

```
1 == 2 && 100 == as.numeric("100")
```

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

```
50 < 51 && as.logical(1) & is.data.frame(mtcars)
```

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

Single `&` versus double `&&`

The single `&` performs comparisons on *all* values.

The double `&&` performs comparisons only until it knows what the outcome will be.

For example, even though the operation below *has* to return `FALSE` (since the first half is `FALSE`), it evaluates the second half anyways (which throws an error)

```
exists("nonexistent_object") & nonexistent_object == 1
```

```
## Error in eval(expr, envir, enclos): object 'nonexistent_object' n
```

Using `&&` will properly return FALSE (because it doesn't evaluate the second half).

```
exists("nonexistent_object") && nonexistent_object == 1
```

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

When given a vector, `&` performs comparisons on all elements:

```
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, TRUE)
```

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

`&&` performs comparisons *only* on the first element:

```
c(TRUE, FALSE, TRUE) && c(TRUE, TRUE, TRUE)
```

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

```
c(FALSE, FALSE, TRUE) && c(TRUE, TRUE, TRUE)
```

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

Bottom line: When using operators to produce one `TRUE`/`FALSE` value, you most likely want to use `&&`

or, or or

The `|` and `||` ('or', 'or or') operators are used to compare whether one of two (or more) conditions are `TRUE`

If one or more conditions `TRUE`, `TRUE` is returned, otherwise `FALSE` (or `NA`) is returned.

```
TRUE || FALSE
```

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

```
FALSE || TRUE
```

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

```
FALSE || FALSE
```

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

Similar to the single `&` and double `&&`, `|` evaluates all conditions while `||` stops when the first `TRUE` is reached

So this throws an error:

```
TRUE | nonexistent_object == 1
```

```
## Error in eval(expr, envir, enclos): object 'nonexistent_object' n
```

But this does not:

```
TRUE || nonexistent_object == 1
```

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

You can combine multiple `||` or evaluate several conditions. For example:

```
FALSE || 1 == 2 || "This" == "That" || TRUE
```

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

When given a vector, `|` performs comparisons on all elements:

```
c(TRUE, FALSE, TRUE) | c(TRUE, TRUE, TRUE)
```

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

`||` performs comparisons *only* on the first element:

```
c(FALSE, FALSE, TRUE) || c(FALSE, TRUE, TRUE)
```

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

not!

Also known as the "bang" operator

Converts `TRUE` into `FALSE` and `FALSE` into `TRUE`

```
!TRUE
```

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

```
!FALSE
```

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

```
# Don't actually do this!
!!!!!!!!!FALSE
```

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

The `!` operator is used when you want to check if a condition does *not* evaluate to `TRUE`. For example to make sure something is not numeric:

```
!is.numeric("ABCD")
```

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

You can negate anything that returns a logical value

```
!(1 == 1 && 2 == 2)
```

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

```
# Function that just returns TRUE when it is called
returnTRUE <- function() return(TRUE)
returnTRUE()
```

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

```
!returnTRUE()
```

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

Type coercion

If a relational or logical operator is passed (i.e., used with) two different atomic vectors as arguments, R will automatically coerce (i.e., change) one type to the other.

Coercion occurs in the following (decreasing) order of precedence:

1. Character
2. Complex
3. Numeric
4. Integer
5. Logical
6. Raw

For example, R will coerce this entire vector (which can only be one atomic type) to character because there is a character inside:

```
c("Psych548", 548.00, 548, TRUE)
```

```
## [1] "Psych548" "548"      "548"      "TRUE"
```

But if we remove the character, it coerces it to numeric:

```
c(548.00, 548, TRUE)
```

```
## [1] 548 548   1
```

This is why operations we looked at previously technically work:

```
"A" > 10
```

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

`10` is coerced to `"10"` and then `"A"` is compared to it. Letters come before numbers in R's character comparison.

```
11 > "10"
```

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

`11` is coerced to `"11"` and the string `"11"` is larger than the string `"10"`

Missing Values

Missing values in R are represented as `NA` (without quotes)

Even one `NA` "poisons the well. Your calculations will return `NA` unless you handle missing values properly:

```
mean(vector_with_NAs)
```

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

```
mean(vector_with_NAs,
     na.rm = TRUE)
```

```
## [1] 3.8
```

The `na.rm` argument in `mean()` removes missing values prior to calculating the mean.

`NA` is technically a logical variable...

```
typeof(NA)
```

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

But it *can* be other types as well (we have missing data for characters and numeric variables too!):

```
as.numeric(NA)
```

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

You can directly tell R which type of `NA` you want to use:

- `NA_real_` (double)
- `NA_integer_` (integer)
- `NA_character_` (character)

```
c(NA_character_, 100)
```

```
## [1] NA    "100"
```

Detecting Missing Values

WARNING: You can't test for missing values by seeing if they are equal to NA (`==NA`):

```
vector_with_NAs == NA
```

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

Instead, you need to use the `is.na()` function:

```
is.na(vector_with_NAs)
```

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

And to check that a value is *not* `NA`:

```
!is.na("This is not NA")
```

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

`Inf` and `NaN`

R also has representations for positive and negative infinity (`Inf`, `-Inf`) and undefined values (`NaN`; **N**ot **a** **N**umber):

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf  NaN  Inf
```

To check if something is finite or not, use `is.finite()`:

```
is.finite(c(-1, 0, 1) / 0)
```

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

To check if something is not defined, use `is.nan()`:

```
is.nan(c(-1, 0, 1) / 0)
```

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

Vectors

Making Vectors

In R, a **vector** is a set of values that are the same atomic type

We create vectors using the `c()` function (for '**c**ombine' or '**c**oncatenate'):

```
c(3, 500, -Inf, -1.23, 24/2*10)
```

```
## [1]   3.00 500.00   -Inf  -1.23 120.00
```

You can also use `:` as shorthand to create vectors of series of numbers (incremented by one):

```
1:10
```

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

```
-5:5
```

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

The following also works but is more unusual to see:

```
1.5:10.5
```

```
##  [1]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
```

You can also create numeric vectors using the `seq()` (for **seq**uence) function:

```
seq(from = 1, to = 11, by = 2)
```

```
## [1]  1  3  5  7  9 11
```

```
seq(from = 1, to = 10, length.out = 5)
```

```
## [1]  1.00  3.25  5.50  7.75 10.00
```

`seq_len` will return the same result as `1:length`:

```
seq_len(4)
```

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

`seq_along` will return a vector of 1:N elements of another vector:

```
seq_along(5:10)
```

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

You can create vectors of repeated values with `rep()`:

```
rep(10, times = 5) # Repeat 10 five times
```

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

```
rep(c(TRUE, FALSE), times = 3) # Repeat c(TRUE, FALSE) 3 times
```

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

```
rep(c("One", "Two"), each = 3) # Repeat each element 3 times
```

```
## [1] "One" "One" "One" "Two" "Two" "Two"
```

Vectors are one-dimensional (**length**) by definition:

```
length(c(90, -0.5, 2, Inf))
```

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

Nested calls to `c()` are flattened:

```
length(c(1, c(2, 3, 4, c(5, 6, 7, 8))))
```

```
## [1] 8
```

If all elements of a vector are not the same type, R will coerce the vector into one type[1]:

```
c("One", 1, TRUE)
```

```
## [1] "One"  "1"    "TRUE"
```

[1] This is the source of many bugs, so be careful with this!

Vector Math

When doing arithmetic operations with vectors, R handles these *elementwise*:

```
# 1*4, 2*5, 3*6
c(1, 2, 3) * c(4, 5, 6)
```

```
## [1]  4 10 18
```

```
# 1^4, 2^4, 3^4, 4^4
c(1, 2, 3, 4)^4
```

```
## [1]   1  16  81 256
```

Other common operations on numeric vectors:

`+`, `-`, `/`, `exp()` = e^x, `log()` = $\log_e(x)$

Vector Recyling

If you do math of vectors with different lengths, R will **recycle** the shorter one by repeating it until it matches the length of the longer one. For example:

```
# 1*1, 2*2, 1*3, 2*4
c(1, 2) * c(1, 2, 3, 4)
```

```
## [1] 1 4 3 8
```

```
# Same exact operation as above
c(1, 2, 1, 2) * c(1, 2, 3, 4)
```

```
## [1] 1 4 3 8
```

You can recycle with a **scalar** (a single number) as well:

```
# 1+1, 1+2, 1+3
1 + c(1, 2, 3)
```

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

Warning on Recycling

R will warn you if you do math with vectors of incommensurate lengths (but it will not throw an error!):

```
# 1+1, 2+2, 3+3, 1+4
c(1, 2, 3) + c(1, 2, 3, 4)
```

```
## Warning in c(1, 2, 3) + c(1, 2, 3, 4): longer object length is no
## shorter object length
```

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

Vectorwise Math

Some functions operate on the entire vector and return one number (rather than operating elementwise):

```
sum(1:5)
```

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

```
max(c(1, 4, 10:999))
```

```
## [1] 999
```

Other vectorwise summary functions include:

`min()`, `mean()`, `median()`, `sd()`, `var()`

Example: Standardizing Data

You can combine elementwise and vectorwise math to perform your calculations. For example, if you want to standardize a vector of values (e.g., scores on a beahvioral tasks):

$$z_i = \frac{x_i - \text{mean}(x)}{\text{SD}(x)}$$

```
scores <- c(30, 28, 47, 27, 97, 49, 84, 78, 33, 48)
z <- (scores - mean(scores)) / sd(scores)
round(z, 2)
```

```
##  [1] -0.87 -0.95 -0.20 -0.99  1.77 -0.12  1.25  1.02 -0.75 -0.16
```

You can also use the built-in `scale()` function to do this for you:

```
identical(z, as.vector(scale(scores)))
```

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

Logical Vectors are Special!

A logical vector is a vector filled with `TRUE` and `FALSE` values

Typically logical vectors are created programmatically as a result of logical tests (e.g., `course == "Psych548"`). For example, if you want to check whether a group of people are old enough to purchase alcohol:

```
ages <- c(13, 43, 72, 24, 21, 20, 40, 15, 29)
ages >= 21
```

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

Logical vectors are most useful when you want to take a subset of a vector (or other data type). Only ages 21+ are selected:

```
ages[ages >= 21]
```

```
## [1] 43 72 24 21 40 29
```

Subsetting Vectors

There are many ways to **subset** a vector. You will primarily use `[` and `]` to specify a subset of a vector, however you can also use the `subset()` function.

```
# Adult clinical core faculty
faculty <- c("Corey", "Angela", "Bill", "Mary", "Jane", "Lori")
```

You can pass a single index (scalar) or vector of values to **keep**:

```
faculty[c(4, 5)]
```

```
## [1] "Mary" "Jane"
```

You can pass a single index or vector or values to **drop**:

```
faculty[-c(1, 3:5)]
```

```
## [1] "Angela" "Lori"
```

You can also pass a logical vector(`TRUE` to keep, `FALSE` to drop):

```
faculty[c(T, F, T, F, F, T)]
```

```
## [1] "Corey" "Bill"  "Lori"
```

Using logical vectors to subset other vectors is incredibly useful

For example, what if the order of the names changed, but you wanted to keep Mary and Jane?

```
faculty == "Mary" | faculty == "Jane"
```

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

```
faculty[faculty == "Mary" | faculty == "Jane"]
```

```
## [1] "Mary" "Jane"
```

Or maybe you want to know which faculty have at least 1 grad student[1]:

```
# How many grad students does each faculty have?
gradstudents <- c(1, 0, 4, 3, 2, 3)
faculty[gradstudents > 1]
```

```
## [1] "Bill" "Mary" "Jane" "Lori"
```

The `subset()` function can also be used to subset vectors:

```
subset(faculty, gradstudents > 1)
```

```
## [1] "Bill" "Mary" "Jane" "Lori"
```

When subsetting vectors, the only difference between `[]` and `subset()` is that `subset()` removes `NA`s

```
myvector <- 1:5
myvector[c(T, T, F, NA, T)]
```

```
## [1]  1  2 NA  5
```

```
subset(myvector, c(T, T, F, NA, T))
```

```
## [1] 1 2 5
```

```
myvector[c(T, T, F, NA, T) & !is.na(c(T, T, F, NA, T))]
```

```
## [1] 1 2 5
```

Named vectors

You can assign **names** to the elements of a vector using the `names()` function:

```
names(gradstudents) <- faculty
print(gradstudents)
```

```
##  Corey Angela   Bill   Mary   Jane   Lori
##      1      0      4      3      2      3
```

☝ The elements of this vector are still numeric!

Names are a useful way of subsetting your data and do not depend on the order of the vector:

```
gradstudents[c("Mary", "Jane")]
```

```
## Mary Jane
##    3    2
```

Helpful Logical/Subsetting Functions

`%in%` allows you to avoid typing a lot of OR (`|`) statements out:

```
# Same as: faculty == "Mary" | faculty == "Jane"
faculty %in% c("Mary", "Jane")
```

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

```
!faculty %in% c("Mary", "Jane") # Faculty not Mary and Jane
```

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

`which()` gives you the indices (locations) of `TRUE` values in a logical vector:

```
which(faculty %in% c("Mary", "Jane"))
```

```
## [1] 4 5
```

Matrices

Making Matrices

Matrices are basically two dimensional vectors with **rows** and **columns** and are made with the `matrix()` function

```
# LETTERS is a built-in vector in R w/ elements A-Z
matrix(LETTERS[1:6], nrow = 2, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,] "A"  "C"  "E"
## [2,] "B"  "D"  "F"
```

The `byrow` argument (defaults to `FALSE`) determines whether the data fills the matrix by row (`TRUE`) or by column (`FALSE`):

```
matrix(LETTERS[1:6], nrow = 2, ncol = 3, byrow = T)
```

```
##      [,1] [,2] [,3]
## [1,] "A"  "B"  "C"
## [2,] "D"  "E"  "F"
```

You can also make matrices by binding vectors together with `rbind()` (**r**ow **bind**) and `cbind()` (**c**olumn **bind**):

```
rbind(1:3, 4:6, 7:9)
```

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

```
cbind(1:3, 4:6, 7:9)
```

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

Let's make a matrix to practice subsetting on the next slide!

```
letters_matrix <- matrix(letters[1:6], nrow = 2,
                         ncol = 3, byrow = T)
print(letters_matrix)
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"
```

Subsetting Matrices

Matrices are subset wtih `[rows, colums]`

```
# Row 2, Column 3
letters_matrix[2, 3]
```

```
## [1] "f"
```

```
# Row 1, Columns 2 and 3
letters_matrix[1, c(2, 3)]
```

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

If you want to keep the entire row or column, keep the space inside the square braces (`[]`) blank:

```
# All rows, column 1
letters_matrix[, 1]
```

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

```
# Row 2, all columns
letters_matrix[2, ]
```

```
## [1] "d" "e" "f"
```

Matrices -> Vectors

When a matrix is subsetted to just 1 row/column of data (like we saw in the previous slide), R will automatically convert it to a vector. You tell R *not* to do this by using `drop = FALSE`:

```
# All rows, column 1
letters_matrix[2, ]
```

```
## [1] "d" "e" "f"
```

```
# Row 2, all columns
letters_matrix[2, , drop = F]
```

```
##      [,1] [,2] [,3]
## [1,] "d"  "e"  "f"
```

To get the dimensions of a matrix, use `dim()`

```
# Returns vector of length 2: c(rows, columns)
dim(letters_matrix)
```

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

Matrix Atomic Type Warning

Like vectors, all elements of a matrix must be the same atomic type. If they are not, R will automatically coerce the matrix according to the rules discussed earlier (character > complex > numeric > integer > logical > raw)

```
cbind(1:2, c("UW", "WSU"))
```

```
##      [,1] [,2]
## [1,] "1"  "UW"
## [2,] "2"  "WSU"
```

Named Matrices

You can assign **names** to rows (`rownames()`) and columns (`colnames()`) of a matrix:

```
mymatrix <- matrix(1:6, nrow = 2)
rownames(mymatrix) <- c("Odds", "Evens")
colnames(mymatrix) <- c("First", "Second", "Third")
print(mymatrix)
```

```
##       First Second Third
## Odds      1      3     5
## Evens     2      4     6
```

You can then subset the matrix by the dimension names:

```
mymatrix["Evens", c("First", "Third"), drop = F]
```

```
##       First Third
## Evens     2     6
```

Matrix Math

If two matrices have the same dimensions, math can be formed elementwise. For example:

```
mat1 <- matrix(1:6, ncol = 3)
print(mat1)
```

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

```
mat2 <- matrix(1:6, ncol = 3, b
print(mat2)
```

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

```
mat1 / mat2
```

```
##      [,1] [,2]     [,3]
## [1,]  1.0  1.5 1.666667
## [2,]  0.5  0.8 1.000000
```

```
mat1 * mat2
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   15
## [2,]    8   20   36
```

Matrix Transposition and Multiplication

To transpose a matrix, use `t()`:

```
mat1t <- t(mat1)
print(mat1t)
```

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

To do actual matrix multiplication (not elementwise), use `%*%`:

```
mat1 %*% mat1t
```

```
##      [,1] [,2]
## [1,]   35   44
## [2,]   44   56
```

Matrix Inversion

To invert an invertible square matrix, use `solve()`:

```
mat4 <- mat1 %*% mat1t
mat4i <- solve(mat4)
print(mat4i)
```

```
##           [,1]      [,2]
## [1,]  2.333333 -1.833333
## [2,] -1.833333  1.458333
```

```
mat4 %*% mat4i
```

```
##              [,1]          [,2]
## [1,] 1.000000e+00 -2.664535e-15
## [2,] 1.776357e-14  1.000000e+00
```

Note the floating point imprecision: The off-diagonals are *very close to zero* rather than actually zero!

Diagonal Matrices

To extract the diagonal of a matrix or make a diagonal matrix (usually the identity matrix), use `diag()`:

```
varcov_mtcars <- cov(mtcars[, 1:3])
print(varcov_mtcars)
```

```
##                mpg        cyl        disp
## mpg     36.324103  -9.172379  -633.0972
## cyl     -9.172379   3.189516   199.6603
## disp -633.097208 199.660282 15360.7998
```

Get the variances of the first 3 variables in `mtcars`:

```
diag(varcov_mtcars)
```

```
##          mpg          cyl         disp
##    36.324103     3.189516 15360.799829
```

You can also use `diag()` to make an identity matrix of size n:

```
diag(2)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
diag(4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

Lists

Lists are objects that can store multiple types of data and are created with `list()`

```
mylist <- list("groceries" = c("Soy Sauce", "Rice", "Tofu"),
               "numbers" = 1:7,
               "mydata" = matrix(8:11, nrow = 2),
               "linearmod" = lm(mpg ~ disp, data = mtcars))
print(mylist)
```

```
## $groceries
## [1] "Soy Sauce" "Rice"      "Tofu"
##
## $numbers
## [1] 1 2 3 4 5 6 7
##
## $mydata
##      [,1] [,2]
## [1,]    8   10
## [2,]    9   11
##
## $linearmod
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Coefficients:
## (Intercept)         disp
##    29.59985     -0.04122
```

Accessing List Elements

You can access a list element by its name or number in `[[ ]]` (note the double square brackets) or `$` followed by its name:

```
mylist[[1]]
```

```
## [1] "Soy Sauce" "Rice"      "Tofu"
```

```
mylist[["groceries"]]
```

```
## [1] "Soy Sauce" "Rice"      "Tofu"
```

```
mylist$groceries
```

```
## [1] "Soy Sauce" "Rice"      "Tofu"
```

Why two brackets `[[ ]]`?

Single brackets return a **list**

Double brackets return the actual list element as whatever data type it is stored as:

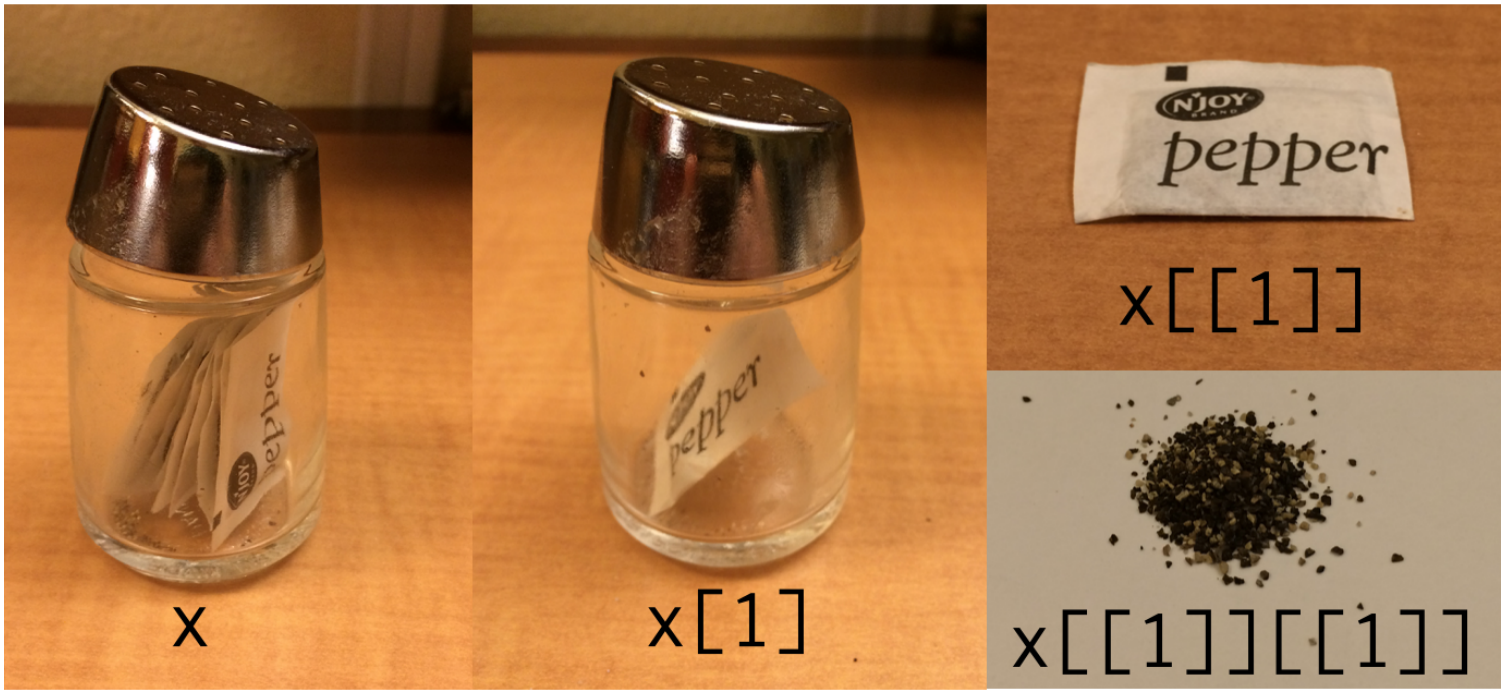
```
typeof(mylist[1])
```

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

```
typeof(mylist[[1]])
```

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

`[]` versus `[[]]`



`[x]` chooses elements but keeps the list while `[[x]]` extracts the element from the list

Source: Hadley Wickham

Regression Output is a List!

```
# Display only the first 7 elements
str(mylist$linearmod, list.len = 7)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 29.5999 -0.0412
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "disp"
##  $ residuals    : Named num [1:32] -2.01 -2.01 -2.35 2.43 3.94 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive
##  $ effects      : Named num [1:32] -113.65 -28.44 -1.79 2.65 3.92 ...
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "disp" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 23 23 25.1 19 14.8 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "disp"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.18 1.09
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##   [list output truncated]
##  - attr(*, "class")= chr "lm"
```

Named Lists

Lists can be unnamed:

```
unnamed_list <- list(c("Apples", "Bananas", "Oranges"),
                     1:10,
                     diag(3))
print(unnamed_list)
```

```
## [[1]]
## [1] "Apples"  "Bananas" "Oranges"
##
## [[2]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[3]]
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

You can use `names()` to access the names of a named list and to assign names to an unnamed list:

```
names(mylist)
```

```
## [1] "groceries" "numbers"   "mydata"    "linearmod"
```

```
names(unnamed_list) <- c("Fruit", "Numbers", "Identity Matrix")
print(unnamed_list)
```

```
## $Fruit
## [1] "Apples"  "Bananas" "Oranges"
##
## $Numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $`Identity Matrix`
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Dataframes

Dataframes are a special type of list where all elements of the list are the same length and are bound together

Unlike matrices, dataframes can hold data of different atomic types (but each column needs to be the same type)

To construct a dataframe from scratch, use the `data.frame()` function:

```
uwclinpsych <- data.frame("name" = c("Corey", "Angela", "Bill", "Ma
                          "grads" = c(1, 0, 4, 3, 2, 3),
                          "fullprof" = c(F, F, T, T, T, T))
print(uwclinpsych)
```

```
##     name grads fullprof
## 1  Corey     1    FALSE
## 2 Angela     0    FALSE
## 3   Bill     4     TRUE
## 4   Mary     3     TRUE
## 5   Jane     2     TRUE
## 6   Lori     3     TRUE
```

You can also create dataframes using `rbind()` and `cbind()`, but unless you are binding two dataframes together, it will return a matrix:

```
cbind("name" = c("Corey", "Angela", "Bill", "Mary", "Jane", "Lori"),
      "grads" = c(1, 0, 4, 3, 2, 3),
      "fullprof" = c(F, F, T, T, T, T))
```

```
##      name     grads fullprof
## [1,] "Corey"  "1"   "FALSE"
## [2,] "Angela" "0"   "FALSE"
## [3,] "Bill"   "4"   "TRUE"
## [4,] "Mary"   "3"   "TRUE"
## [5,] "Jane"   "2"   "TRUE"
## [6,] "Lori"   "3"   "TRUE"
```

--

```
rbind(uwclinpsych[4:6, ], uwclinpsych[1:3, ])
```

```
##     name grads fullprof
## 4   Mary     3     TRUE
## 5   Jane     2     TRUE
## 6   Lori     3     TRUE
## 1  Corey     1    FALSE
## 2 Angela     0    FALSE
## 3   Bill     4     TRUE
```

Subsetting Dataframes

Dataframes are subset in the same way as matrices(`[rows, columns]`)

```
uwclinpsych[, 1]
```

```
## [1] "Corey"  "Angela" "Bill"   "Mary"   "Jane"   "Lori"
```

```
uwclinpsych[, "name"]
```

```
## [1] "Corey"  "Angela" "Bill"   "Mary"   "Jane"   "Lori"
```

```
uwclinpsych[c(1, 3, 5), ]
```

```
##    name grads fullprof
## 1 Corey     1    FALSE
## 3  Bill     4     TRUE
## 5  Jane     2     TRUE
```

You can also use the `$` operator to target a single column

```
uwclinpsych$name
```

```
## [1] "Corey"  "Angela" "Bill"   "Mary"   "Jane"   "Lori"
```

When you have data that are related to each other and it is possible to store them as a dataframe, you should! This allows you to confidently make subsets for your analyses:

```
uwclinpsych[uwclinpsych$grads > 2, "name"]
```

```
## [1] "Bill" "Mary" "Lori"
```

👇 same return value

```
uwclinpsych$name[uwclinpsych$grads > 2]
```

```
## [1] "Bill" "Mary" "Lori"
```

```
uwclinpsych$name[uwclinpsych$grads > 2 &
                   uwclinpsych$fullprof]
```

☝ What is this code doing?

```
## [1] "Bill" "Mary" "Lori"
```

If you want to subset one column of a dataframe while keeping it as a dataframe object, use `drop=FALSE`:

```
uwclinpsych[, 1]
```

```
## [1] "Corey"  "Angela" "Bill"   "Mary"   "Jane"   "Lori"
```

```
uwclinpsych[, 1, drop = FALSE]
```

```
##     name
## 1  Corey
## 2 Angela
## 3   Bill
## 4   Mary
## 5   Jane
## 6   Lori
```

Viewing Dataframes

You can view an entire dataframe by `print()`ing it out in the console.[1] However, it often just spams your console and is too large to meaningfully read.

To view the first *n* rows of a dataframe use `head()` (defaults to 5 rows):

```
head(uwclinpsych, 3)
```

```
##     name grads fullprof
## 1  Corey     1    FALSE
## 2 Angela     0    FALSE
## 3   Bill     4     TRUE
```

To view the last *n* rows of a dataframe use `tail()` (defaults to 5 rows):

```
tail(uwclinpsych, 4)
```

```
##   name grads fullprof
## 3 Bill     4     TRUE
## 4 Mary     3     TRUE
## 5 Jane     2     TRUE
## 6 Lori     3     TRUE
```

[1] You can also type the name of the dataframe without `print()` and R will print it!

You can also `View()` a more friendly pop-up of your data and, in RStudio, filter and sort as you view

```
View(swiss)
```

	Fertility	Agriculture	Examination	Education	Catholic	Infant.Mortality
Courtelary	80.2	17.0	15	12	9.96	22.2
Delemont	83.1	45.1	6	9	84.84	22.2
Franches-Mnt	92.5	39.7	5	5	93.40	20.2
Moutier	85.8	36.5	12	7	33.77	20.3
Neuveville	76.9	43.5	17	15	5.16	20.6
Porrentruy	76.1	35.3	9	7	90.57	26.6
Broye	83.8	70.2	16	7	92.85	23.6
Glane	92.4	67.8	14	8	97.16	24.9
Gruyere	82.4	53.3	12	7	97.67	21.0
Sarine	82.9	45.2	16	13	91.38	24.4
Veveyse	87.1	64.5	14	6	98.61	24.5
Aigle	64.1	62.0	21	12	8.52	16.5
Aubonne	66.9	67.5	14	7	2.27	19.1

Attributes

Objects in R can have **attributes**, which are basically metadata that describe an object. To get an object's attributes, use the `attributes()` function.

This can help give you an overview of an object's properties. For example:

```
# Linear regression w/ one predictor
linearmod <- lm(mpg ~ disp, data = mtcars)
attributes(linearmod)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

Attributes also offer a nice way to document a dataframe

Consider the following data:

```
print(dat)
```

```
##    pid txm phq pcl
## 1 1001   1  10   3
## 2 1002   2  17  28
## 3 1003   3   3  43
## 4 1004   1  25  17
```

🤔 what is `pid`?

🤔 what is `txm`?

🤔 what is `phq`?

🤔 what is `pcl`?

We can assign attributes to each column to document what each is:

```
attr(dat$pid, "Label") <- "Participant ID"
attr(dat$txm, "Label") <- "Treatment Modality"
attr(dat$phq, "Label") <- "Depression (measured by PHQ-9)"
attr(dat$pcl, "Label") <- "PTSD Sx (measured by PCL-5)"

# Columns can have as many attributes as you would like!
attr(dat$txm, "Values") <- c("1" = "CPT", "2" = "PE", "3" = "TAU")
```

You and your collaborators can then reference these attributes

```
attributes(dat)
```

```
## $names
## [1] "pid" "txm" "phq" "pcl"
##
## $row.names
## [1] 1 2 3 4
##
## $class
## [1] "data.frame"
```

```
attributes(dat$txm)
```

```
## $Label
## [1] "Treatment Modality"
##
## $Values
##     1     2     3
## "CPT"  "PE" "TAU"
```