mpr— title: “General R (Reading and Writing Images)” output: ioslides_presentation: widescreen: yes css: ../styles.css beamer_presentation: default bibliography: ../refs.bib
—

R Basics

Data Classes

- Numeric - numbers (e.g. 1, 3.673)
- Character - strings or words (`"hey"` , `"I'm a string"`) in either single or double quotes
- Logicals - `TRUE` or `FALSE` - all capital letters and are **not** in quotes.

Data Types

- `vector` - 1-dimensional object of one class (all numeric or all character)
- `matrix` - 2-dimensional object of one class
- `data.frame` - 2-dimensional object, can be multiple classes (like Excel spreadsheet)
- `array` - object of > 2 dimensions of one class. The data in a `nifti` object is one of these (usually 3-D)

Initializing: vectors

- Use `c()` to create a vector of numeric values:

```
v = c(1, 4, 3, 7, 8)
print(v)
```

```
[1] 1 4 3 7 8
```

- Shortcut for sequential numeric vector:

```
w = 1:5
print(w)
```

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

The gray boxes denote code, and the lines after denote the output.

Assignment

In `R`, you can assign using the equals `=` or arrow `<-` (aka assigment operator).

The above commands are equivalent to:

```
w = 1:5
w <- 1:5
```

There are no differences in these 2 commands, but just preference (we use `=`).

Variable/object names:

- must start with a letter
- cannot contain spaces, `$`, quotes, or other special characters
    - generally just alpha-numeric
- **can** contain periods (`.`) and underscores `_`

Help

- To see the documentation for a function, use the `?` symbol before the name of the function. This is a shortcut for the `help` command:
- For example, to see documentation for `c`:

```
?c
help(topic = "c")
```

- To search for help files, use a double `??` or `help.search`:

```
??c
help.search(pattern = "c")
```

Some Details

- R is case sensitive (e.g. `y` and `Y` are different)
- Commands separated by new line or by a colon (;)
- Use `#` to comment

You can also be explicit about which package you are using with the `::` operator, where the syntax is `package::function()`:

```
utils::help("c")
```

Initializing: matrices and arrays

- Create a 3 x 4 numeric matrix and assign to variable `m`
    - note items are added column-wise

```
m = matrix(1:12, nrow = 3)
print(m)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

- Create a 3 x 4 x 3 numeric array and assign to variable `a`

```
a = array(1:36, dim = c(3, 4, 3))
```

- the `dim()` function returns the dimensions of the array

```
dim(a)
```

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

Subsetting: vectors

- Subsetting a vector (first index is `1`, not zero):

```
print(v)
```

```
[1] 1 4 3 7 8
```

```
print(v[4])
```

```
[1] 7
```

```
print(v[1:3])
```

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

```
print(v[c(1,3,5)])
```

```
[1] 1 3 8
```

Subsetting: matrices

- Subsetting a matrix - `[row,column]` format,

```
print(m[1,3])
```

```
[1] 7
```

```
print(m[1:2,3:4])
```

```
     [,1] [,2]
[1,]    7   10
[2,]    8   11
```

- if `row` or `column` missing then all values printed:

```
print(m[,4])
```

```
[1] 10 11 12
```

```
print(m[2,])
```

```
[1]  2  5  8 11
```

Subsetting: arrays

- Subsetting - `[x,y,z]` format:

```
print(a[1,1,1])
```

```
[1] 1
```

```
dim(a[,4,])
```

```
[1] 3 3
```

This will return an error - need to specify all dims:

```
a[,4]
```

Operators in R: return numeric

- Arithmetic: `+`, `-`, `*`, `/`, `^` - exponents
- Standard math functions: `log`, `abs`, `sqrt`

```
print(v); print(w)
```

```
[1] 1 4 3 7 8
```

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

```
print(v + 4)
```

```
[1]  5  8  7 11 12
```

```
print(v + w)
```

```
[1]  2  6  6 11 13
```

```
print(sqrt(w^2))
```

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

Operators in R: return logical

- Comparison: `>` , `>=` , `<` , `<=` , `==` (equals), `!=` (not equal)
- Logical: `!` - not, `&` - and, `|` - or (a “pipe”)
- `all()` : function to test all values `TRUE` and `any()` : (are any)

```
print(!FALSE)
```

```
[1] TRUE
```

```
print(TRUE | FALSE)
```

```
[1] TRUE
```

```
print(FALSE & FALSE)
```

```
[1] FALSE
```

```
c(all(c(TRUE, FALSE)), any(c(TRUE, FALSE)))
```

```
[1] FALSE  TRUE
```

Subsetting with logicals

The `which` command takes a logical and gets the indices of `TRUE` :

```
which(v > 5)
```

```
[1] 4 5
```

```
v[ which(v > 5) ]
```

```
[1] 7 8
```

Or directly pass in a vector of logicals to subset:

```
v[ v > 5 ]
```

```
[1] 7 8
```

This method will be useful later when we are working with images.

Imaging Packages in R

Some packages we will use

All packages we will discuss are loaded on the RStudio Server:

- `oro.nifti` - reading/writing NIfTI images
    - made the `nifti` object/data class: like an array - but with header information
        - the main data class we will use
- `neurobase` - extends `oro.nifti` and provides helpful imaging functions

Let's load them:

```
library(oro.nifti)
library(neurobase)
```

Reading in NIfTI images: assignment

We will use the `readnii` function (from `neurobase`) to read in a `nifti` object (this is an `R` object).

Here we read in the "training01_01_t1.nii.gz" file, and assign it to an object called `t1`:

```
t1 = readnii("training01_01_t1.nii.gz")
```

Now, an object `t1` is in memory/the workspace.

```
class(t1)
```

```
[1] "nifti"
attr(,"package")
[1] "oro.nifti"
```

`nifti` images

By default, if you simply pass the object, it is printed, we can also do `print(t1)` :

```
t1
```

```
NIfTI-1 format
  Type            : nifti
  Data Type       : 4 (INT16)
  Bits per Pixel  : 16
  Slice Code      : 0 (Unknown)
  Intent Code     : 0 (None)
  Qform Code      : 2 (Aligned_Anat)
  Sform Code      : 1 (Scanner_Anat)
  Dimension       : 408 x 512 x 152
  Pixel Dimension : 0.43 x 0.43 x 0.82
  Voxel Units     : mm
  Time Units      : Unknown
```

Operations with `nifti` objects

These work with an image and a number (`img + 2`) or two images of the same dimensions `img1 + img2` .

- Comparison: `>` , `>=` , `<` , `<=` , `==` (equals), `!=` (not equal)
- Logical: `!` - not, `&` - and, `|` - or (a “pipe”)
- Arithmetic: `+` , `-` , `*` , `/` , `^` - exponents
- Standard math functions: `log` , `abs` , `sqrt`

```
t1 + t1 + 2 # still a nifti
```

```
NIfTI-1 format
  Type            : nifti
  Data Type       : 4 (INT16)
  Bits per Pixel  : 16
  Slice Code      : 0 (Unknown)
  Intent Code     : 0 (None)
  Qform Code      : 2 (Aligned_Anat)
  Sform Code      : 1 (Scanner_Anat)
  Dimension       : 408 x 512 x 152
  Pixel Dimension : 0.43 x 0.43 x 0.82
  Voxel Units     : mm
  Time Units      : Unknown
```

Working with `nifti` objects

Again, we can use a logical operation. Let's create an image indicating values over 400:

```
class(t1 > 400) # still a nifti
```

```
[1] "nifti"
attr(,"package")
[1] "oro.nifti"
```

```
head(t1 > 400) # values are now logical vs. numeric
```

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

We will refer to images such as `t1 > 400` as a "mask", simply binary images with logical values in them (or 0s and 1s)

Subsetting with `nifti` objects: like `arrays`

The subsetting here is similar to that of arrays. Since `t1` is 3-dimensional the subsetting goes to the 3rd dimension:

```
t1[5, 4, 3]
```

```
[1] 0
```

```
t1[5, 4, ] # returns a vector of numbers (1-d)
t1[, 4, ] # returns a 2-d matrix
t1[1, , ] # returns a 2-d matrix
```

- You can subset with a logical array of the same dimensions!
- We can view values of the `t1` greater than 400 (`head` only prints the first 6 values):

```
head(t1[ t1 > 400 ]) # produces a vector of numbers
```

```
[1] 402 412 435 448 453 430
```

`which` with `nifti` objects

The `which` function works to get indices, but you can pass the `arr.ind = TRUE` argument to get "array" indices:

```
head(which(t1 > 400, arr.ind = TRUE))
```

```
     dim1 dim2 dim3
[1,]  180  258    1
[2,]  175  259    1
[3,]  176  259    1
[4,]  177  259    1
[5,]  178  259    1
[6,]  179  259    1
```

But can get the "vector" indices as well:

```
head(which(t1 > 400, arr.ind = FALSE))
```

```
[1] 105036 105439 105440 105441 105442 105443
```

Working with `nifti` objects: reassignment

Subsetting can work on the left hand side of assignment too:

```
t1_copy = t1
t1_copy[ t1_copy > 400 ] = 400 # replaced these values!
max(t1_copy) # should be 400
```

```
[1] 400
```

```
max(t1)
```

```
[1] 1691
```

Note, although `t1_copy` was copied from `t1`, they are not linked - if you change values in `t1_copy`, values in `t1` are unchanged.

Writing Images out

We now can write out this modified `t1_copy` image:

```
writenii(nim = t1_copy,
         filename = "training01_t1_under400.nii.gz")
file.exists("training01_t1_under400.nii.gz")
```

```
[1] TRUE
```

We have seen that `file.exists` returns `TRUE` if a file exists

- useful in conjunction with `all`: `all(file.exists(VECTOR_OF_FILES))`

Vectorizing a `nifti`

To convert a `nifti` to a `vector`, you can simply use the `c()` function:

```
vals = c(t1)
class(vals)
```

```
[1] "numeric"
```

Essentially “strings out” the array. If you do `array(c(t1), dim = dim(t1))`, this will put things back “in order” of the `t1`.

Vectorizing is useful for making `data.frame`s (covered later) when you want to do modeling at a voxel level.

```
df = data.frame(t1 = c(t1), mask = c(t1 > 400)); head(df)
```

```
  t1  mask
1  0 FALSE
2  0 FALSE
3  0 FALSE
4  0 FALSE
5  0 FALSE
6  0 FALSE
```

File helpers - for constructing filenames

Use `paste` if you want to put strings together with spaces, `paste0` no spaces by default.

`file.path(directory, filename)` will paste `directory` and `filename` w/file separators (e.g. `/`)

```
c(paste("img", ".nii.gz"), paste0("img", ".nii.gz"))
```

```
[1] "img .nii.gz" "img.nii.gz"
```

```
x = file.path("output_directory", paste0("img", ".nii.gz")); print(x)
```

```
[1] "output_directory/img.nii.gz"
```

`nii.stub` will strip off the nifti extension. If `bn = TRUE`, it removes the directory as well:

```
c(nii.stub(x), nii.stub(x, bn = TRUE))
```

```
[1] "output_directory/img" "img"
```

Main Packages we will use

- `oro.nifti` - reading/writing NIfTI images
- `neurobase` - extends `oro.nifti` and provides helpful imaging functions
- `fslr` - wraps FSL commands to use in R

    - registration, image manipulation, skull stripping
- `ANTsR` - wrapper for Advanced normalization tools (ANTs) code
    - registration, inhomogeneity correction, lots of tools
- `extrantsr` - allows `ANTsR` to work with objects from `oro.nifti`

Data Package we will use

- `ms.lesion` - contains training/testing data of patients with multiple sclerosis (MS)
    - from the MS lesion challenge 2016 (http://iacl.ece.jhu.edu/index.php/MSChallenge (http://iacl.ece.jhu.edu/index.php/MSChallenge))

Conclusions

- We have (briefly) covered some R data classes and types to get you started
- We will be using `nifti` objects
    - They are special 3-dimensional arrays
    - Contain numbers or logicals
- `readnii` and `writenii` are used for reading/writing `nifti` objects to NIfTI files
- We have briefly covered subsetting and image manipulation
    - more on that later

Website

http://johnmuschelli.com/imaging_in_r (../index.html)