Bio 724: Foundations of Scientific Computing

Paul M. Magwene

Goals for today's class session

- Make sure you have a working R/RStudio environment
- Make sure you know how to evaluate commands in the R terminal
- Make sure you know how to create and knit an RMarkdown document
- Review R data types – numerical (double, integer, complex), logical, character
- Review R data structures – vectors, lists, data frames

RStudio Tour

In-class demonstration of RStudio interface.

RMarkdown

In-class demonstration of creating and knitting an RMarkdown document.

Data types

- Refers to the types of values that can be represented in a computer program
- Determine the representation of values in memory
- Constrains the operations you can perform on those values

In R data types are usually inferred by the interpreter rather than specified by the user, though occasionally you may find it necessary to specify a data type.

Numeric data types in R

- Floating point values ('doubles') – represent real numbers (continuous values)

```
> x <- 10.0
> typeof(x)
[1] "double"
```

- Integers – represent whole numbers. The default numeric type is double, so you must explicitly ask for integers.

```
> x <- as.integer(10)
> typeof(x)
[1] "integer"
```

- Complex numbers – numbers with a real and "imaginary" part. We won't be explicitly using complex numbers in this course, but they sometimes appear unexpectedly in some calculations.

```
> x <- 1+1i
> typeof(x)
[1] "complex"
```

Arithmetic operations on numerical data types

```
> 10 + 2 # addition
[1] 12
> 10 - 2 # subtraction
[1] 8
> 10 * 2 # multiplication
[1] 20
> 10 / 2 # division
[1] 5
> 10 ^ 2 # exponentiation
[1] 100
> 10 ** 2 # alternate exponentiation
[1] 100
> 10 %% 4 # modulus (remainder after division)
[1] 2
> 4-5 / 2 # operator precedence matters!
[1] 1.5
> (4-5)/2 # parentheses help you specify/disambiguate precedence.
[1] -0.5
```

Basic mathematical functions

```
> sqrt(10) # square root
[1] 3.162278
> 10 ^ 0.5 # same as square root
[1] 3.162278
> sqrt(-1) # NaN means "Not a Number"
[1] NaN
> sqrt(-1 + 0i) # But works if we use complex type
[1] 0+1i
> exp(1) # exponential function
[1] 2.718282
> log(100)  # log base e
[1] 4.60517
> log10(100) # log base 10
[1] 2
> log2(8) # log base 2
[1] 3
> factorial(5) # factorial function: 5 * 4 * 3 * 3 * 1
[1] 120
> pi  # R knows some useful constants
[1] 3.141593
> cos(2*pi)  # cosine, also sin, tan, ...
[1] 1
```

Logical (Boolean) type

```
> x <- TRUE
> typeof(x)
[1] "logical"
> y <- FALSE
> typeof(FALSE)
[1] "logical"
> !TRUE # logical negation of TRUE
[1] FALSE
> !y # logical negation (NOT) of value in y
[1] TRUE
```

Numerical comparison operators return logical types:

```
> 4 < 5  # less than
[1] TRUE
> 10 >= 9 # greater than or equal to
[1] TRUE
```

Comparison operations can be assigned to variables:

```
> x <- 1 > 2
> y <- 2 >= 2
> x & y # Logical AND
[1] FALSE
> x | y # Logical OR
[1] TRUE
> isTRUE(x) # returns true if x is logical and true
[1] FALSE
```

Character (string) type

```
> x <- "Hello" # enclosed in double quotes
> y <- 'World' # or single quotes
> z <- 'You said "Hello World"' # allows nesting
```

Simple character functions:

```
> paste(x, y)  # concatenate strings
[1] "Hello World"
> paste(x, y, sep = "") # concatenate with no space
[1] "HelloWorld"
> strsplit("Hello world!", split=" ") # split on space
[[1]]
[1] "Hello"  "world!"
```

The `stringr` package contains many useful string manipulation functions.

Data structures

- Represent different ways collections of data are stored in memory or accessed by the user
- Homogenous or heterogeneous with respect to data types
- Different data structures are more efficient for particular modes of access or to represent different types operations

Vectors

- Homogeneous ordered list of items

```
> x <- c(2, 4, 6, 8)
> y <- c("hello", "world", "how", "are", "you?")
```

Vectors always have a length (possibly zero) and a type

```
> length(x)
[1] 4
> typeof(x)
[1] "double"
> length(y)
[1] 5
> typeof(y)
[1] "character"
```

Elements of a vector will be coerced to be the same type

```
> x <- c(1+1i, 2+1i, 'Fred', 10)
> x
[1] "1+1i" "2+1i" "Fred" "10"
> typeof(x)
[1] "character"
```

Indexing vectors

Accessing the objects in a vector is accomplished by "indexing".

The elements of the vector are assigned indices $1 \ldots n$ where n is the length of the vector

```
> x <- c(2,4,6,8)
> length(x)
[1] 4
> x[1]
[1] 2
> x[2]
[1] 4
> x[4]
[1] 8
> x[length(x)] # why might this be preferred way to get last value?
[1] 8
```

Indexing past the end of a vector returns `NA` (Not Available) values:

```
> x[5]
[1] NA
```

Single objects of core data types in R are themselves vectors

```
> x <- 1
> length(x)
[1] 1
> x[1] # can be indexed like any other vectors
[1] 1
> is.vector(x)  # function to test whether something is vector
[1] TRUE
```

Arithmetic operators and most math functions work on numerical vectors

Arithmetic operators and comparison work element-by-element on vectors.

```
> x <- c(2, 4, 6, 8)
> y <- c(0, 1, 2, 10)
> x + y
[1]  2  5  8 18
> x * y
[1]  0  4 12 80
> x^2
[1]  4 16 36 64
> sqrt(x)
[1] 1.414214 2.000000 2.449490 2.828427
> x < y
[1] FALSE FALSE FALSE  TRUE
```

Vector Recycling

When binary operators are applied to vectors of different lengths, "recycling" occurs - the shorter vector is repeated to match the length of the longer one by recycling its elements starting from the first one:

```
> x <- c(1, 1, 1, 1)
> y <- c(0, 1)
>
>          # original       after recylcing    result
>          # x = 1 1 1 1    x = 1 1 1 1        1 2 1 2
>          # y = 0 1        y = 0 1 0 1
> x + y
[1] 1 2 1 2
```

Vectors can be indexed by other vectors, including logical vectors

```
> x <- c(2, 4, 6, 8)
> x[c(2,4)]  # get 2nd and 4th elements of x
[1] 4 8
```

```
> x <- c(2, 4, 6, 8)
> y <- c(0, 1, 2, 10)
> x[x > y] # get elements of x where x > y
[1] 2 4 6
```

Lists

Lists in R are like vectors but the elements of a list are arbitrary objects (even other lists). Lists are "heteregeneous".

```
> x <- list('Bob',27, 10, c(720,710))
> typeof(x)
[1] "list"
> x
[[1]]
[1] "Bob"

[[2]]
[1] 27

[[3]]
[1] 10

[[4]]
[1] 720 710
```

Indexing Lists

Items in lists are accessed in a different manner than vectors.

- Typically you use double brackets (`[[]]`)to return the element at index `i`
- Single brackets always return a list containing the element at index `i`

```
> x <- list('Bob', 27, 10, c(720,710))
> x[1]
[[1]]
[1] "Bob"
> typeof(x[1])
[1] "list"
> x[[1]]
[1] "Bob"
> typeof(x[[1]])
[1] "character"
```

List elements can have names

```
> x <- list(name='Bob',age=27, years.in.school=10)
> x
$name
[1] "Bob"

$age
[1] 27

$years.in.school
[1] 10
```

Named list objects can be accessed via the $ operator

```
> x$years.in.school
[1] 10
```

The names of list elements can be accessed with the `names()` function

```
> names(x)
[1] "name"            "age"             "years.in.school"
```

Data frames

Data frames represent data tables (data sets)

- Each column in the table has the same number of rows
- Every item in a given column has to be of the same type.
- Columns in a data frame must have names

```
> # constructing a data frame from scratch
> name <- c("Paul","Mary","Peter")
> grade <- c("C","A","B")
> age <- c(40, 42, 28)
>
> example.df <- data.frame(name = name, age = age, grade = grade)
> example.df
   name age grade
1  Paul  40     C
2  Mary  42     A
3 Peter  28     B
```

Data frames: shape and column indexing

Shape

```
> nrow(example.df)  # number of rows
[1] 3
> ncol(example.df)  # number of columns
[1] 3
```

Indexing columns by name or position

```
> example.df$age
[1] 40 42 28
> example.df[3]
  grade
1     C
2     A
3     B
```

Indexing with a vector to subset columns

```
> example.df[c("name","grade")]
   name grade
1  Paul     C
2  Mary     A
3 Peter     B
```

Data frames: indexing rows

To index one or more rows of a data frame, specifying the value(s) to index on followed by a comma:

```
> example.df[1,]  # first row
  name age grade
1 Paul  40     C
```

```
> example.df[c(1,3),] # first and third row
   name age grade
1  Paul  40     C
3 Peter  28     B
```

You can simultaneously index both rows and columns:

```
> example.df[c(1,3), c("name", "age")]
   name age
1  Paul  40
3 Peter  28
```

Matrices

In R matrices are two-dimensional collections of elements all of which have the same mode or type (homogenous).

```
> r <- matrix(1:12, nrow=3, ncol = 4)
> r
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Matrices don't by default have explicit row or column names, but they can be assigned/retrieved using the `rownames()` and `colnames()` functions respectively.

```
> colnames(r) <- c("N", "C", "O2", "PO4")
> rownames(r) <- c("Site.A", "Site.B", "Site.C")
> r
       N C O2 PO4
Site.A 1 4  7  10
Site.B 2 5  8  11
Site.C 3 6  9  12
```

We will typically prefer data frames to matrices, because data frames are more flexible. But we'll return to matrices when we get into linear algebra later in the semester.