ILLINOIS

R Data Types

Lecture 9

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)

Department of Statistics, University of Illinois

ILLINOIS

Outline

- Vector, Matrix, ... of `int`, `double`, `char`, `logical`, ...
- `NA`, `NaN`, `NULL`
- Factors
- `Date`, `Datetime`
- Types, dispatch, classes, ...
- `data.frame`, `list`

Code Examples

Type the code in the console, hit RETURN and confirm the result:

```
R> 2^3
# [1] 8
R>
R> 0:4
# [1] 0 1 2 3 4
R>
R> (0:4)^0.5
# [1] 0.00000 1.00000 1.41421 1.73205 2.00000
```

Key Points:

- Expressions can be on scalars, or vectors
- Standard rules of operator precedence apply:
  - `(0:5)^2` is different from `0:5^2`
  - using parantheses to give priorities
- Display is “smart” and suppresses information past the decimal point when uninformative *but results still at full precision*

Code Example

```
R> a <- pi
R> a
# [1] 3.14159
R> a <- 42L
R> a
# [1] 42
R> a <- "The quick brown fox"
R> a
# [1] "The quick brown fox"
```

Key Points:

- R is a “dynamically-typed” language:
  - Variable are dynamically typed
  - *i.e.* each assignment sets the type
  - previous type does not matter
  - good for interactive exploration
  - possible source of bugs in programming
- Variables do not need to be declared

More Key Points:

- all R types are vectors
  - in fact, there is no “scalar” in R
  - *everything* is a vector
  - sometimes of length one
- vectors can be combined (more on that below)

Three Basic Numeric Types

- `numeric` is `double` (also `real`): "numbers with decimal point"
- `integer`: "whole number" *i.e.* no decimal point
- also `complex` but we rarely encounter it in statistics
- smart conversion when needed from `integer` to `real`
- most of the time we use `numeric`

Internal Representation

- The storage mode for floating point is `double`
- The storage mode for whole numbers is `integer`
  - These take up 64 and 32 bits, respectively.
  - But as R programmers we rarely need to think of the bit details.
- But knowing a little about double / floating point is helpful.
  - See 7.31 Why doesn't R think these numbers are equal?
  - This is from the (overall excellent) R FAQ
  - And is such a classic that people sometimes just mutter "7.31" ...

Very Useful

- R extends the usual floating point standard and supports
  - `NaN` is not a number: something unrepresentable such `0/0`
  - `NA` is not available: something missing (useful for data work)
  - `NULL` is unknown: yet another state but subtly different from `NA`
- `numeric` also has `Inf` and `-Inf`
- Special values are (generally) also available for other types
  - R is unique in having `NA` for integer
  - Other systems generally only have `NaN` and `NA` for floats
  - SQL generally only has `NULL`

Testing

- `is.null()` tests for `NULL`
- `is.nan()` tests for `NaN`
- `is.finite()` and `is.finite()` useful too:

```
R> is.finite(c(NaN, NA, NULL, Inf, -Inf))
# [1] FALSE FALSE FALSE FALSE
R> is.infinite(c(NaN, NA, NULL, Inf, -Inf))
# [1] FALSE FALSE  TRUE  TRUE
```

Other Types

- `char` for character variables, also supporting `NA`
- `logical` for Boolean `TRUE` or `FALSE`
  - but also `NA` so three-valued (!!)
- `raw` for storage / network transmission which we rarely need
- `factor` for limited dependent variables

Why Factors?

- Good for modeling and *programming with data*
- A little unusual at first but very useful
- Encodes 'limited-depedent' variables as
  - an internal integer value indexing
  - `as.integer()` or `as.numeric()` extracs the values
  - a string vector with labels
  - `levels()` extracts that vector
  - `as.character()` converts the factor into character vector

```
R> data(iris, package="datasets")
R> lm(Sepal.Length ~ Species - 1, data=iris)
#
# Call:
# lm(formula = Sepal.Length ~ Species - 1, data = iris)
#
# Coefficients:
#     Speciessetosa  Speciesversicolor   Speciesvirginica
#              5.01               5.94               6.59
```

Exercise: Do you know other ways to compute conditional means?

Date

- `Date` is supported as a class built on top of `double`
  - *i.e.* fractional days are supported
- `Sys.Date()` generates current date
- Try this: `as.Date("2019-09-20") + 0:4`

Datetime

- `Datetime` is support via
  - `POSIXct`:
    - a compact representation of fractional seconds
    - relative to the "epoch", ie Jan 1, 1970
    - try `Sys.time()` and `as.double(Sys.time())`
  - `POSIXlt`:
    - a list representation of year, mon, mday, ... components
  - `POSIXct` and `POSIXlt` are easy convert back and forth
    - Both inherit from `POSIXt`

Many helpful functions

- `difftime()`, see units argument; also with `as.double()`
- `seq()` on Date or Datetime objects
- `strftime()` to format
- `strptime()` to parse
- `as.POSIXct()` to convert to compact
- `as.POSIXlt()` to convert to list
- `weekday()` and other extractors

Types drive beahvior

- R dispatches on type for so-called *generic* functions
- If you invoke a method, say, `print()` with arguments
- R will (generally) look at the first argument of your call and determine its type
- If it is of `type`, and if corresponding method `print.type()` exists, it will be called
- Otherwise `print.default` is called (and every generic has to have a default)

Custom Types

- "S Programming", Section 4.1, by Venables and Ripley (2000) has a very nice worked example on a custom hypothesis tests
- In essence:
  - define `Ttest()` generic with `UseMethod("Ttest")`
  - define `Ttest.default()` as default method:
    - doing computation of test
    - setting class `"my.t.test"`
  - define `print.my.t.test()`
- Calling `Ttest(x1, x2)` does the work *and* dispatches to custom print method

Basics

- Everything is a vector, though sometimes lenth one
- Matrices are implemented as a vector *with a dimension attribute*
- Both vectors and matrices are stored internally as one contiguous memory chunk
- But matrix indices use two dimension
- Conversion between matrix and vector pretty seamless
- But matrix attribute can get lost when vector extracted
- Argument `drop=FALSE` can ensure matrix type persists
- Try this and reason about it:

```
m <- matrix(1:9,3,3); m[,1]; m[,1,drop=FALSE]
```

- Vectors and matrix *can* also have (row|col)names

Indexing

- By index position: `vec[c(3,5,12)]`
- By logical value: `vec[c(TRUE,FALSE,FALSE,TRUE)]`
- By name: `vec[c("tic", "tac")]` (if `vec` has names)
- By expression: `vec[ someExpressionHere ]`
  - where the expression yields one of the three earlier types
- Special case: `m[n]` where `m` and `n` are matrices
  - Try this and have a guess before you run it:

```
m <- matrix(1:16,4); n <- matrix(1:4,2); m[n]
```

Arrays are multidimensional vectors

- Vectors can have `dim` attributes with more than two dimension
- Not all that frequently used
- More awkward `print()` etc functions
- But worth knowing these exist

Lists are a catch-all data structure

- Lists are the only data structure that
  - can contain elements of *different length*
  - can be nested: list containing list containing...
- Lists frequently used internally for *implementation*
  - with a simple S3 class use around them
  - so `print()`, `summary()`, etc for finer control
- We will not have time for that latter aspect

DataFrames are a very popular and widely-used type

- Rectangular: Rows are observations, columns are variables
- Internally a list of vectors
- All vectors *must be* of same length
- Each vector of one type only
- But different type vectors permitted
- Ideal container for data sets and modeling
- Powerful idea – now been ported from R to Python, Julia, ...

DataFrames are a very popular and widely-used type

- A `data.frame` object will always have column names
- But row names are optional
- The `stringsAsFactor=TRUE` default irks some people
    - keep it in mind when reading/constructing data.frames
- Some types extend `data.frame`, we will see this later
- Indexing by row/col index, or by (partial) match on column:
    - `iris[3, "Species"]` is preferred;

      expressions for row or col index permitted, very general
    - `iris$Species[3]` also work but not recommended

Indexing

- By position: `df[rowexpression, columnexpression]`
- This is very general:
  - simplest case is scalar: `iris[2, 3]`
  - missing means all: `iris[4, ]` or `iris[, 2:3]`
  - variables and expression can be used
  - columns can be index by position or name
  - logical indexing: `iris[ iris[,2] > cutoff, ]`
  - or by name: `iris[ iris[, "Sepal.Width"] > 4, ]`
- This can be wordy and later we will see better alternatives

‘Bread and Butter’ types and operations we learned about:

- Vectors of `integer`, `numeric`, `character`, ...
- Often used as columns in a `data.frame`
- And/or parts of a list
- We learned about `NA`, `NaN` and `NULL`
- We considered different indexing methods