R Basics

James McDermott

University of Galway


Programming and Tools for
Artificial Intelligence
Dr James McDermott

R Basics

Basic R

R **syntax** looks a bit different to Python. Many people think it's not as clean. But the fundamental **concepts** are mostly the same: line-by-line execution, primitive data types, compound data types, assignment, function calls, iteration and conditionals, classes.

Numerical data

```
x <- 5
typeof(x) # vector of length 1 of type double!
```

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

```
x <- 5L
typeof(x) # using 'L', a vector of length 1 of type int
```

```
## [1] "integer"
```

```r
s <- "abcdefghi" # assignment using "<-"
x <- 5
while (x > 0) { # curly brackets
  if (x %% 2 == 0) { # operators may differ from Python
    print(c(x, substr(s, x, x))) # c() means concatenate
  }
  x = x - 1 # assignment using "="
}
```

```
## [1] "4" "d"
## [1] "2" "b"
```

Special values

- `NA`: not applicable/missing (common in data read from e.g. CSV files)
- `NaN`: "not a number", as in Python
- `-Inf`, `Inf`: infinite values, as in Python

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

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

Special functions

`is.finite()`, `is.infinite()`, `is.na()`, `is.nan()`

What's this about <- and =?

They usually do the same thing – assignment – but there are a few places where = is not allowed. The R community tends to stick to <-.

- https://stackoverflow.com/questions/1741820/what-are-the-differences-between-and-in-r

Factors

A *factor* is a primitive datatype for categorical data. It's especially useful in statistical formulae.

```
x <- factor(c("cat", "dog", "cat"),
            levels=c("mouse", "cat", "dog"))
typeof(x) # integer?! yes, behind the scenes
```

```
## [1] "integer"
```

```
length(x)
```

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

Vectors: a compound data type

```
x <- c("one", "two", "three", "four", "five")
x[1] # BTW R indexes from 1, not from 0
```

```
## [1] "one"
```

```
x[[1]]
```

```
## [1] "one"
```

What's the difference between [] and [[]]?

Both exist and sometimes do the same thing, sometimes different!

```
x <- c("one", "two", "three", "four", "five")
x[1] # BTW R indexes from 1, not from 0
```

```
## [1] "one"
```

```
x[[1]]
```

```
## [1] "one"
```

```
x[c(3, 2, 5)] # single [] for selecting a subset of elements
```

```
## [1] "three" "two"   "five"
```

```
# x[[c(3, 2, 5)]] # double [[]] doesn't work here
```

- https://stackoverflow.com/questions/1169456/the-difference-between-bracket-and-double-bracket-for-accessing-the-el
- http://adv-r.had.co.nz/Subsetting.html

Lists with named elements

A `list` with named elements is a bit like a `dict`:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
a[["b"]] # double square brackets
```

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

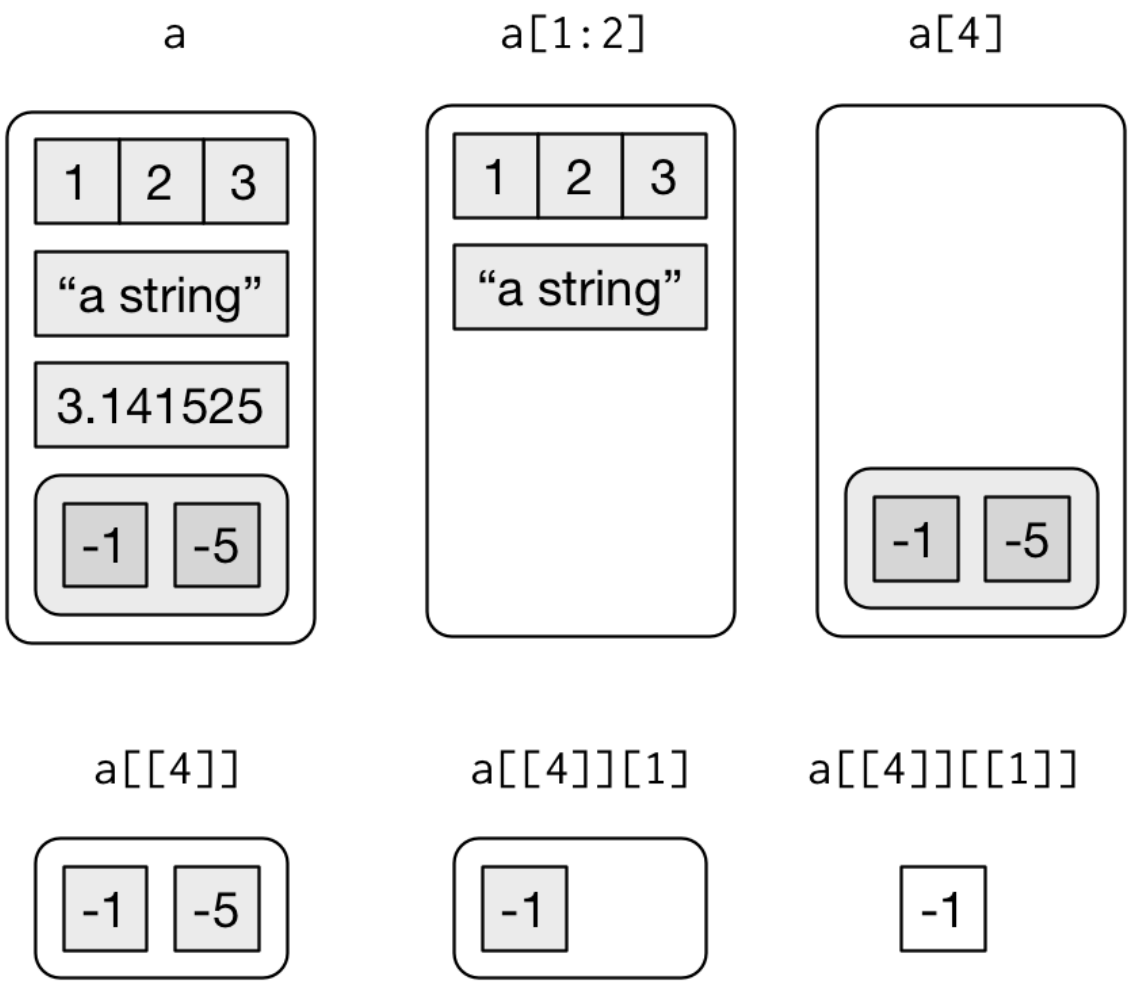
`$` is the same as `[[]]` but only for named elements:

```
a$b # notice, no quotation marks
```

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

List subsetting

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```



(From R4DS)

List subsetting

See also Wickham's pepper pot: https://r4ds.had.co.nz/vectors.html (Ctrl-F pepper)

Compound data types

- R `vector` -> Python `list` or Numpy `ndarray`
- R `list` -> Python `tuple`
- R named `list` -> Python `dict`
- R `data.frame` -> Pandas `DataFrame`
- R `tibble` -> Pandas `DataFrame`

Inspecting compound data types

The `str` function gives you the *structure* of an item:

```
str(a)
```

```
## List of 4
##  $ a: int [1:3] 1 2 3
##  $ b: chr "a string"
##  $ c: num 3.14
##  $ d:List of 2
##   ..$ : num -1
##   ..$ : num -5
```

The `typeof` and `length` functions are self-explanatory:

```
typeof(a)
```

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

```
length(a)
```

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

Ranges, columns, for-in

```
xs = 1:10 # range 1, 2, ... 10
print(xs)
```

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

```
for (x in xs) {
  print(x^2 %% 2)
}
```

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

But don't use for-loops!

R is vectorised, like Numpy:

```
xs = 1:10
xs = xs ^ 2
ys = xs %% 2
ys
```

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

Vectorised if-else

```
v1 = 1:5
v2 = v1 ^ 2
v3 = ifelse(v2 %% 2 == 0, "Even", "Odd")
v3
```

```
## [1] "Odd"  "Even" "Odd"  "Even" "Odd"
```

Compare to Numpy `np.where`.

Recycling

```
1:10 * 1:2
```

```
##  [1]  1  4  3  8  5 12  7 16  9 20
```

The shorter vector is recycled. But this is ugly: don't use it!

Functions

`function` is the equivalent of Python `lambda`.

```
normalise <- function(x) {
  # no "return" statement: last value is returned
  (x - min(x)) / (max(x) - min(x))
}
normalise(1:10)
```

```
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.55
##  [8] 0.7777778 0.8888889 1.0000000
```

Exercises

1. Write the Factorial function in R, eg `fact(5)` gives 120.
2. Given `x <- "John"`, calculate the length in characters of `x`. Use `nchar()`.
3. Given `xs <- c("John", "Paul", "George", "Ringo")`, calculate the length of each name, using vectorisation (not a for-loop).
4. Calculate whether each name is shorter than 5 characters.
5. Index `xs` to keep just the names shorter than 5 characters.
6. Write a function which unit-norms a vector, ie normalises it so that the vector length equals 1. Eg `unit_norm(c(10, 10, 10, 10))` gives `0.5 0.5 0.5 0.5`.
7. Write a function which standardises a vector, ie gets the z-score, ie maps it to have mean 0 and standard deviation 1. Eg `z_score(c(10, 6, 12, 12))` gives `0.0000000 -1.4142136  0.7071068 0.7071068`.

Solutions

```
fact <- function(n) { # Exercise 1
  if (n <= 1) {
    1 # remember, no return statement!
  } else {
    n * fact(n-1)
  }
}
fact(5)
```

```
## [1] 120
```

```r
x <- "John"
nchar(x) # Exercise 2
```

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

```r
xs <- c("John", "Paul", "George", "Ringo")
nchar(xs) # Exercise 3
```

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

```r
nchar(xs) < 5 # Exercise 4
```

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

```r
xs[nchar(xs) < 5] # Exercise 5
```

```
## [1] "John" "Paul"
```

```r
unit_norm <- function(x) { # Exercise 6
  x / sqrt(sum(x**2))
}
unit_norm(c(10, 10, 10, 10))
```

```
## [1] 0.5 0.5 0.5 0.5
```

```r
z_score <- function(x) { # Exercise 7
  (x - mean(x)) / sd(x)
}
z_score(c(10, 6, 12, 12))
```

```
## [1]  0.0000000 -1.4142136  0.7071068  0.7071068
```