R Fundamentals

First Encounter

First Steps

- R as a calculator — show the numerical result of $1 + 2 \times \frac{3}{4} - 5 + 6 \sin \frac{\pi}{2} - \sqrt[8]{7}$

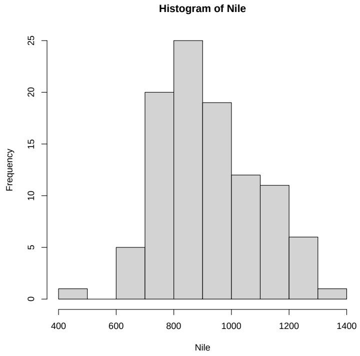
```
1 + 2 * (3 / 4) - 5 + 6 * sin(pi / 2) - 7^(1/8)
# [1] 2.224627
```

- Print the content of the built-in dataset `Nile`

```
Nile
# Time Series:
# Start = 1871
# End = 1970
# Frequency = 1
#   [1] 1120 1160  963 1210 1160 1160  813 1230 1370 1140  995  935 1110  994 1020
#  [16]  960 1180  799  958 1140 1100 1210 1150 1250 1260 1220 1030 1100  774  840
#  [31]  874  694  940  833  701  916  692 1020 1050  969  831  726  456  824  702
#  [46] 1120 1100  832  764  821  768  845  864  862  698  845  744  796 1040  759
#  [61]  781  865  845  944  984  897  822 1010  771  676  649  846  812  742  801
#  [76] 1040  860  874  848  890  744  749  838 1050  918  986  797  923  975  815
#  [91] 1020  906  901 1170  912  746  919  718  714  740
```

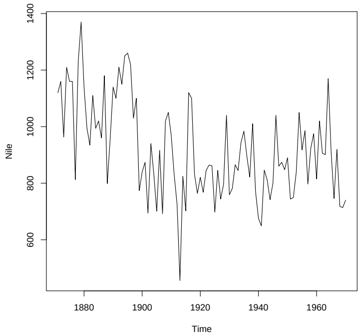
- Show the histogram of `Nile`

```
hist(Nile)
```



- Show the plot of `Nile`

```
plot(Nile)
```



Vectors and Indices

- The 2nd item of the `Nile` dataset

```
Nile[2]
# [1] 1160
```

- Subsequence taken from 2nd to 5th item of `Nile`

```
Nile[2:5]
# [1] 1160  963 1210 1160
```

- Subsequence taken from 2nd to 5th item of `Nile`: using the concatenate function `c`

```
Nile[c(2, 3, 4, 5)]
# [1] 1160  963 1210 1160
```

- Subsequence taken from 10th, 3rd, 15th item of `Nile`

```
Nile[c(10, 3, 15)]
# [1] 1140  963 1020
```

- Let `x` be a vector `(10, 20, -1, 3, 5, 6, 30, 9)`

```
(x <- c(10, 20, -1, 3, 5, 6, 30, 9))
# [1] 10 20 -1  3  5  6 30  9
```

- Let `y` be a vector formed by taking the 2nd element out of `x`

```
(y <- x[-2])
# [1] 10 -1  3  5  6 30  9
```

- Let `z` be a vector formed by taking the 2nd, 3rd, 6th element out of `x`

```
(z <- x[c(-2, -3, -6)])
# [1] 10  3  5 30  9
```

- Let `w` be the vector formed by combining `x, y, z`

```
(w <- c(x, y, z))
#  [1] 10 20 -1  3  5  6 30  9 10 -1  3  5  6 30  9 10  3  5 30  9
```

- Subsequence taken from 81th to 95th item of `Nile`:

```
Nile[81:95]
#  [1]  744  749  838 1050  918  986  797  923  975  815 1020  906  901 1170  912
```

- Lengths of the subsequence `Nile[81:95]`

```
length(Nile[81:95])
# [1] 15
```

- Let the subsequence `Nile[81:95]` be `v8195`; Compute its mean and standard deviation

```
v8195 <- Nile[81:95]; mean(v8195); sd(v8195)
# [1] 913.6
# [1] 116.4645
```

- Let the vector `x <- c(20, 1, 15, 13, 12)`. Print `x > 14`:

```r
x <- c(20, 1, 15, 13, 12); x > 14
# [1]  TRUE FALSE  TRUE FALSE FALSE
```

- How many items in `x` that >14 ?

```r
sum(x > 14)
# [1] 2
```

- Which of the items in `Nile` are >1200 ?

```r
which(Nile > 1200)
# [1]  4  8  9 22 24 25 26
```

- Print out the items in `Nile` that >1200.

```r
Nile[Nile > 1200]
# [1] 1210 1230 1370 1210 1250 1260 1220
```

Data Frame

- data frame: a rectangular table consisting of one row for each data point. Built-in data frame `ToothGrowth`, assigned `tg`.

- len: tooth length; supp: supplement VC (vitamin c) or OJ (orange juice); dose

```
head(ToothGrowth)
```

```
#    len supp dose
# 1  4.2   VC  0.5
# 2 11.5   VC  0.5
# 3  7.3   VC  0.5
# 4  5.8   VC  0.5
# 5  6.4   VC  0.5
# 6 10.0   VC  0.5
```

- Each column is a vector; use `$` to extract

```
tg <- ToothGrowth; tg$len
```

```
#  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 16.5 16.5 15.2 17.3 22.5
# [16] 17.3 13.6 14.5 18.8 15.5 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5 23.3 29.5
# [31] 15.2 21.5 17.6  9.7 14.5 10.0  8.2  9.4 16.5  9.7 19.7 23.3 23.6 26.4 20.0
# [46] 25.2 25.8 21.2 14.5 27.3 25.5 26.4 22.4 24.5 24.8 30.9 26.4 27.3 29.4 23.0
```

- To get the item of `tg` in row 3, column 1

```
tg[3, 1]
# [1] 7.3
```

- To get the item of `tg` in row 3, column 1: using that `tg$len` is a vector

```
tg$len[3]
# [1] 7.3
```

- Extract rows 2 through 5, and columns 1 and 3, assigning the result to `z`

```
(z <- tg[2:5, c(1, 3)])
#    len dose
# 2 11.5  0.5
# 3  7.3  0.5
# 4  5.8  0.5
# 5  6.4  0.5
```

- To get the items of `tg` in columns 1, 3

```r
head(tg[, c(1, 3)])
```

```
#    len dose
# 1  4.2  0.5
# 2 11.5  0.5
# 3  7.3  0.5
# 4  5.8  0.5
# 5  6.4  0.5
# 6 10.0  0.5
```

- To get the items of `tg` in columns 1, 3: remove column 2

```r
head(tg[, -2])
```

```
#    len dose
# 1  4.2  0.5
# 2 11.5  0.5
# 3  7.3  0.5
# 4  5.8  0.5
# 5  6.4  0.5
# 6 10.0  0.5
```

- Create your own data frame from vectors of same lengths

```r
age <- c(55, 58, 45)
name <- c('Alice', 'Bill', 'Cathy')
(d <- data.frame(age, name))
```

```
#   age  name
# 1  55 Alice
# 2  58  Bill
# 3  45 Cathy
```

- Note that vector of different lengths are not allowed !

```r
blood <- c('O', 'B', 'A', 'AB')
data.frame(age, name, blood)
```

```
# Error in data.frame(age, name,
blood): arguments imply differing
number of rows: 3, 4
```

Extracting Rows / Columns from Data Frames

- In tg, compute mean tooth length for supplement VC and OJ

```
tg_vc <- tg[tg$supp == 'VC',]
tg_oj <- tg[tg$supp == 'OJ',]
mean(tg_vc$len)
# [1] 16.96333
mean(tg_oj$len)
# [1] 20.66333
```

- Extract the sub data frame from tg with supplement OJ and length < 8.8

```
tg[tg$supp == 'OJ' & tg$len < 8.8,]
#    len supp dose
# 37 8.2   OJ  0.5
```

- Note the existence and the postion of the comma (,) and the use of == !

- Extract the sub data frame from tg with length > 28 or dose $= 1.0$

```
w <- tg[tg$len == 28 | tg$dose == 1.0,]
head(w, 13)
#     len supp dose
# 11 16.5   VC    1
# 12 16.5   VC    1
# 13 15.2   VC    1
# 14 17.3   VC    1
# 15 22.5   VC    1
# 16 17.3   VC    1
# 17 13.6   VC    1
# 18 14.5   VC    1
# 19 18.8   VC    1
# 20 15.5   VC    1
# 41 19.7   OJ    1
# 42 23.3   OJ    1
# 43 23.6   OJ    1
```

List

- In `mtcars`, split the miles-per-gallon (mpg) data according to the number of cylinders (cyl):

```
(mtl <- split(mtcars$mpg, mtcars$cyl))
```

```
# $`4`
#  [1] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26.0 30.4 21.4
#
# $`6`
# [1] 21.0 21.0 21.4 18.1 19.2 17.8 19.7
#
# $`8`
#  [1] 18.7 14.3 16.4 17.3 15.2 10.4 10.4 14.7 15.5 15.2 13.3 19.2 15.8 15.0
```

```
class(mtl)
```

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

- Now vectors in `mtl`, a list, can be accessed individually with `` $` ` `` and `[[ ]]`:

```
mtl$`6`; mtl[[2]]
```

```
# [1] 21.0 21.0 21.4 18.1 19.2 17.8 19.7
# [1] 21.0 21.0 21.4 18.1 19.2 17.8 19.7
```

- In R, data frames can't hold vectors of different lengths, but lists can.

```
vn <- c(1, 1.2, 2.3, 3.4, 4.5)
vb <- c(TRUE, TRUE, FALSE)
vc <- c('limestone', 'marl', 'oolite', 'CaCO3')
```

```
data.frame(vn, vb, vc)
```

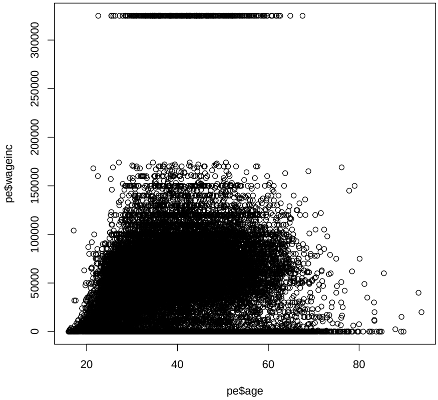
```
# Error in data.frame(vn, vb, vc): arguments imply differing number of rows:
5, 3, 4
```

```
list(vn, vb, vc)
```

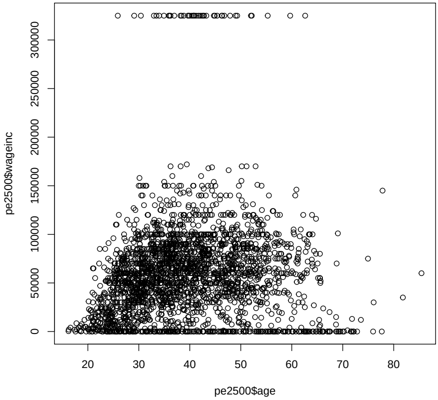
```
# [[1]]
# [1] 1.0 1.2 2.3 3.4 4.5
#
# [[2]]
# [1]  TRUE  TRUE FALSE
#
# [[3]]
# [1] "limestone" "marl"      "oolite"    "CaCO3"
```

Graphics

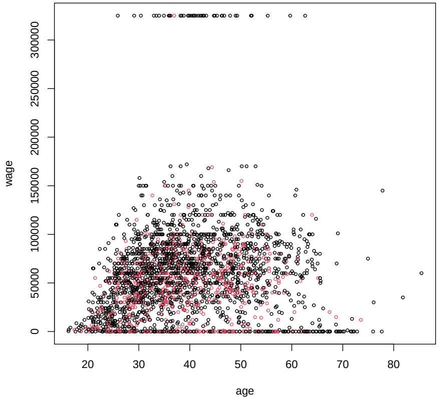
```
pe <- read.table(
    'https://raw.githubusercontent.com/matloff/fasteR/master/data/prgeng.txt',
    header=TRUE)
plot(pe$age, pe$wageinc)
```



```r
pe2500 <- pe[sample(1:nrow(pe), 2500),]
plot(pe2500$age, pe2500$wageinc)
```



```r
plot(pe2500$age, pe2500$wageinc,
     col=as.factor(pe2500$sex), xlab='age', ylab='wage', cex=0.6)
```



- We have been using built-in functions such as `mean`, `var`.

```
v <- c(1, 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8)
cat('mean: ', mean(v), 'var: ', var(v))
# mean:  4.0625 var:  6.37125
```

- Now write our own:

```
my_mean <- function(x) {
    return(sum(x) / length(x))
}
my_mean(v)
# [1] 4.0625
```

```
my_var <- function(x) {
    m <- my_mean(x)
    return(sum((x - m)^2 / (length(x) - 1)))
}
my_var(v)
# [1] 6.37125
```

- How to sum 1 to 10000 if we don't use built-in functions?

```
ans <- 0
for (i in 1:10000) {
    ans <- ans + i
}
ans
```

```
# [1] 50005000
```

- How to sum $\sum_{k=1}^{100} \min\{2^k, k^4\}$?

```
ans <- 0
for (k in 1:100) {
    ans <- ans + min(2^k, k^4)
}
ans
```

```
# [1] 2050220551
```

- Bonus: Do you know how to do $\sum_{k=1}^{100} \min\{2^k, k^4\}$ quickly without loops?

```
v = 100; sum(pmin(2^v, v^4))
```

- How to sum a vector `x` if we don't use built-in functions?

```
my_sum <- function(x) {
    ans <- 0
    for (i in 1:length(x)) {
        ans <- ans + x[i]
    }
    ans
}
my_sum(1:100)
```

```
# [1] 5050
```

- But it's SOOOO inefficient comparing to built-in functions ...

```
v <- 1:1000000; system.time(my_sum(v)); system.time(sum(v))
```

```
#    user  system elapsed
#   0.016   0.000   0.017
#    user  system elapsed
#       0       0       0
```

- Your Turn: Write a function `myprod` to replicate the built-in function `prod`.

- conditional: `if`, `else`

```
v <- c(20, 18, 45, 33, 25, 9, 17, 55, 69, 81, 44, 32); nv <- length(v)
w <- vector(length = nv)
for (i in 1:nv) {
    if (v[i] >= 20 & v[i] < 40) {
        w[i] <- 0
    } else if (v[i] >= 40 & v[i] < 60) {
        w[i] <- -1
    } else {
        w[i] <- v[i]
    }
}
v; w
#  [1] 20 18 45 33 25  9 17 55 69 81 44 32
#  [1]  0 18 -1  0  0  9 17 -1 69 81 -1  0
```

- vector version: `ifelse`

```
ww <- ifelse(v >= 20 & v < 40, 0, v)
ww <- ifelse(ww >= 40 & ww < 60, -1, ww); ww
#  [1]  0 18 -1  0  0  9 17 -1 69 81 -1  0
```

Misc: Linear Regression, Help Functions

- Linear regression model

```r
head(cars)
model <- lm(dist ~ speed, data = cars)
print(model)
summary(model)
plot(model)
abline(model)
```

- built-in help mechanisms; functions source lookup

```r
?prod
example(cumsum)
example(persp)
help.search('multivariate normal')
help(mgcv::rmvn)
help(rmvn)
rmvn
mgcv::rmvn
prod
var
sd
```

Operators

<table>
<tr><th>Operators</th><th>Definition</th></tr>
<tr><td><code>+ - * /</code></td><td>plus, minus, times, divide</td></tr>
<tr><td><code>%/% %% ^</code></td><td>integer quotient, modulo, power</td></tr>
<tr><td><code>> >= < <=</code></td><td>greater than, greater than or equals, less than, less than or equals</td></tr>
<tr><td><code>== !=</code></td><td>equals, not equals</td></tr>
<tr><td><code>! & |</code></td><td>not, and, or</td></tr>
<tr><td><code><- -></code></td><td>assignment (gets)</td></tr>
<tr><td><code>$</code></td><td>list indexing ('the element name' operator)</td></tr>
<tr><td><code>:</code></td><td>sequence creation</td></tr>
<tr><td><code>~</code></td><td>model formulae ('is modelled as a function of')</td></tr>
<tr><td><code>%x%</code></td><td>special binary operators: <code>x</code> can be set by any valid name</td></tr>
<tr><td><code>%*%</code></td><td>special binary operator: matrix product</td></tr>
<tr><td><code>%in%</code></td><td>special binary operator: test matching</td></tr>
<tr><td><code>&& ||</code></td><td>vector and, or</td></tr>
<tr><td><code>::</code></td><td>namespace assignment</td></tr>
<tr><td><code>xor(a, b)</code></td><td>elementwise exclusive OR</td></tr>
<tr><td><code><<-</code></td><td>global assignment</td></tr>
</table>

Functions

Function	Definition
`abs(x)`	absolute value of `x`
`floor(x)`	greatest integer less than `x`
`ceiling(x)`	smallest integer greater than `x`
`trunc(x)`	closest integer to `x` between `x` and `0`
`round(x, digits = 0)`	round the value of `x` into integer
`signif(x, digits = 6)`	give `x` to 6 significant digits in scientific notation
`runif(n)`	generate `n` random numbers from uniform$(0, 1)$
`choose(n, m)`	binomial coefficient $\binom{n}{m}$
`log(x)`	log to base e of `x`
`log(x, n)`	log to base `n` of `x`
`log10(x)`	log to base 10 of `x`
`exp(x)`	$\exp(x)$
`sqrt(x)`	$\sqrt{x}$
`factorial(x)`	$x! = x \times (x-1) \times (x-2) \times \cdots \times 2 \times 1$
`sin(x), cos(x), tan(x)`	$\sin x$, $\cos x$, $\tan x$

Function	Definition
`max(x)`	maximum value in `x`
`min(x)`	mininum value in `x`
`sum(x)`	sum of all the values in `x`
`mean(x)`	arithmetic average of the values in `x`
`median(x)`	median value in `x`
`quantile(x, p)`	vector corresponding to the given probability `p` of `x`
`range(x)`	vector of `min(x)` and `max(x)`
`rank(x)`	vector of the ranks of the values in `x`
`order(x)`	an integer vector containing the permutation to sort `x` into asc order
`var(x)`	sample variance of `x`
`cor(x, y)`	correlation between vectors `x` and `y`.
`sort(x)`	a sorted copy of `x`
`cumsum(x)`	vector containing the sum of all the elements up to that point
`cumprod(x)`	vector containing the product of all the elements up to that point
`pmax(x, y, z)`	vector containing the maximum of `x, y, z` at each position
`pmin(x, y, z)`	vector containing the minimum of `x, y, z` at each position
`colMeans(x)`	column means of dataframe or matrix `x`
`colSums(x)`	column totals of dataframe or matrix `x`
`rowMeans(x)`	row means of dataframe or matrix `x`
`rowSums(x)`	row totals of dataframe or matrix `x`

Vectors

- flavors: atomic, generic (lists)
- primary types of atomic vector: logical, integer, double, character; integer + double = numeric
- esoteric types of atomic vector: complex, raw
- scalars: vectors of length 1
    - logical: `TRUE`, `FALSE`, `T`, `F`
    - double: `0.1234`, `1.23e4`, `0xcafe`, `NaN`, `-Inf`, `Inf`
    - integer: `1234L`, `1e4L`, `0xcafeL`
    - character: `"hi"`, `'bye'`, special characters escaped with `\`; see `?Quotes` for details.
- demonstrations

```
v_logical <- c(TRUE, FALSE) # atomic vector: logical
v_integer <- c(1L, 6L, 10L) # atomic vector: integer
v_double <- c(1, 2.5, 4.5)  # atomic vector: double
v_character <- c('these', 'are', 'characters') # atomic vector: character
# concatenation of atomic vectors yield atomic vectors
(v <- c(c(1, 2), c(3, 4)))
```

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

```
typeof(v_logical); typeof(v_integer); typeof(v_double); typeof(v_character)
```

```
# [1] "logical"
# [1] "integer"
# [1] "double"
# [1] "character"
```

- missing values: NA (Not Applicable); be existent but unknown.

```r
NA > 5; 20 * NA; !NA  # NA tend to be infectious!
# [1] NA
# [1] NA
# [1] NA
NA ^ 0; NA | TRUE; NA & FALSE  # some exceptions
# [1] 1
# [1] TRUE
# [1] FALSE
x <- c(NA, 5, NA, 10); x == NA; is.na(x) # using is.na() to test
# [1] NA NA NA NA
# [1]  TRUE FALSE  TRUE FALSE
```

- NULL: nonexistent value

```r
u <- NULL; length(u)
# [1] 0
v <- NA; length(v)
# [1] 1
```

- useful tests: is.null(), is.na()
- testing vector type: use is.logical(), is.integer(), is.double(), is.character()
- testing vector type: avoid is.vector(), is.atomic(), is.numeric()

- R feature: automatic coercion of vector elements; `as.*()` to coerce

```r
typeof(c('a', 1))
# [1] "character"
x <- c(TRUE, FALSE, TRUE, TRUE); as.numeric(x)
# [1] 1 0 1 1
as.integer(c('1', '1.5', 'a'))
# Warning: NAs introduced by coercion
# [1]  1  1 NA
```

- type coercion hierarchy: character > double > integer > logical

```r
c(1, FALSE); c(TRUE, 1L); 1 == '1'; -1 < FALSE; 'one' < 2
# [1] 1 0
# [1] 1 1
# [1] TRUE
# [1] TRUE
# [1] FALSE
```

- `:` generating consecutive sequence

```r
5:10; 5:1; i <- 5; 1:i-1; 1:(i-1)
# [1]  5  6  7  8  9 10
# [1] 5 4 3 2 1
# [1] 0 1 2 3 4
# [1] 1 2 3 4
```

- seq — generalized :

```
seq(from = 11, to = 30, by = 3); seq(from = 1.1, to = 2, length = 10)
# [1] 11 14 17 20 23 26 29
#  [1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

- using seq in for-loops

```
for (i in 1:length(x)) {}  # What if length(x) == 0 ?
for (i in seq(x)) {}       # Use this instead !
(x <- c(9, 5, 17)); seq(x); (x <- NULL); seq(x)
```

- The recycling rule: in operations involving two vectors, R repeats the shorter one

```
c(2, 4, 3) + c(6, 10, -1, 3, 5)
# Warning in c(2, 4, 3) + c(6, 10, -1, 3, 5): longer object length is not a
multiple of shorter object length
# [1]  8 14  2  5  9
c(2, 4, 3, 2, 4) + c(6, 10, -1, 3, 5)
# [1]  8 14  2  5  9
c(2, 4, 3) * c(6, 10, -1, 3, 5)
# Warning in c(2, 4, 3) * c(6, 10, -1, 3, 5): longer object length is not a
multiple of shorter object length
# [1] 12 40 -3  6 20
c(2, 4, 3, 2, 4) * c(6, 10, -1, 3, 5)
# [1] 12 40 -3  6 20
```

- example: how to select even terms in a vector?

```
x <- 1:10; x[x %% 2 == 0]
# [1]  2  4  6  8 10
x <- sin(1:10); x[seq(x) %% 2 == 0]; x[c(F, T)]
# [1]  0.9092974 -0.7568025 -0.2794155  0.9893582 -0.5440211
# [1]  0.9092974 -0.7568025 -0.2794155  0.9893582 -0.5440211
```

- `rep`: repeating vector constants

```
(x <- rep(3, 5)); rep(c(10, 15, 3), 4); rep(5:1, 3)
# [1] 3 3 3 3 3
#  [1] 10 15  3 10 15  3 10 15  3 10 15  3
#  [1] 5 4 3 2 1 5 4 3 2 1 5 4 3 2 1
```

- some other examples

```
rep(c(10, 15, 3), each = 4) # interleave the copies with `each`
#  [1] 10 10 10 10 15 15 15 15  3  3  3  3
rep(1:4, each = 2, times = 3); rep(1:4, 1:4); rep(1:4, c(4, 1, 4, 2))
#  [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
#  [1] 1 2 2 3 3 3 4 4 4 4
#  [1] 1 1 1 1 2 3 3 3 3 4 4
rep(c('C', 'G', 'H'), c(4, 5, 2))
#  [1] "C" "C" "C" "C" "G" "G" "G" "G" "G" "H" "H"
```

- ```r
  rep(1:4, each = c(4, 3, 2))
  # Warning in rep(1:4, each = c(4, 3, 2)): first element used of 'each'
  argument
  #  [1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
  ```

- `all` and `any`

  ```r
  x <- 1:100; any(x > 200); all(x > 0)
  # [1] FALSE
  # [1] TRUE
  ```

- testing vector equality: not `==`

  ```r
  x <- 1:3; y0 <- c(1, 2, 3); y1 <- c(1, 3, 4); x == y0; x == y1
  # [1] TRUE TRUE TRUE
  # [1]  TRUE FALSE FALSE
  ```

- consider `identical` and `all.equal` — use `isTRUE(all.equal(…))` in `if` expr !

  ```r
  identical(x, y0); typeof(x); typeof(y0)
  # [1] FALSE
  # [1] "integer"
  # [1] "double"
  all.equal(x, y0); all.equal(x, y1)
  # [1] TRUE
  # [1] "Mean relative difference: 0.4"
  ```
  ```

- `attr()`, `attributes()`, `structure()`: make other data structures from atomic vectors by adding attributes

```
a <- 1:3; attr(a, 'x') <- 'ab'; attr(a, 'x')
# [1] "ab"
attr(a, 'y') <- 4:6; str(attributes(a))
# List of 2
#  $ x: chr "ab"
#  $ y: int [1:3] 4 5 6
a <- structure(1:3, x = 'ab', y = 4:6); str(attributes(a))
# List of 2
#  $ x: chr "ab"
#  $ y: int [1:3] 4 5 6
```

- 3 ways to name a vector `x`

```
x <- c(a = 1, b = 2, c = 3)
x <- 1:3; names(x) <- c('a', 'b', 'c')
x <- setNames(1:3, c('a', 'b', 'c'))
```

- adding a `dim` attribute to vectors make 2d matrices and nd arrays

```
x <- matrix(1:6, nrow = 2, ncol = 3)  # 2d matrix
x1 <- 1:6; dim(x1) <- c(2, 3); x1
y <- array(1:12, c(2, 3, 2))  # 3d array
y1 <- 1:12; dim(y1) <- c(2, 3, 2); y1
```

- S3 atomic vectors in Base-R
  - categorical data: fixed set of levels in `factor` vectors
  - date (with day resolution): recorded in `Date` vectors
  - date-time (with second / subsecond resolution): recorded in `POSIXct` vectors
  - durations: stored in `difftime` vectors
- factors are used when we have categorical variables.

```
(x <- factor(c('a', 'b', 'b', 'a'))); str(x); typeof(x); attributes(x);
```

- `tapply`: simple example

```
ages <- c(25, 26, 55, 37, 21, 42); affils <- c('R', 'D', 'D', 'R', 'U', 'D')
tapply(ages, affils, mean)
```

- `tapply`: further example

```
d <- data.frame(list(gender = c('M', 'M', 'F', 'M', 'F', 'F'),
                     age = c(47, 59, 21, 32, 33, 24),
                     income = c(55000, 88000, 32450, 76500, 123000, 45650)))
d$over25 <- ifelse(d$age > 25, 1, 0)
tapply(d$income, list(d$gender, d$over25), mean)
```

Common Tasks

- create a vector from given values: c

```r
# Note the coercion hierarchy: character, double, integer, logical
v1 <- c(1, 2, 3); v2 <- c('a', 'b', 'c'); (v3 <- c(v1, v2))
```

```
[1] "1" "2" "3" "a" "b" "c"
```

- comparison of two vectors: element-by-element

```r
# Note the vector recycling (and scalars are vectors of length 1)!
v <- c(4, pi); w <- c(pi, pi, pi, 3);
v == w; v != w; v < w; v <= w; v > w; v >= w; w != pi
```

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

- comparison of two vectors: all at once

```r
v <- c(3, pi, 4); any(v == pi); all(v == pi)
```

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

- selecting vector elements by numerical vectors (including scalars)

```
fib <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144);
fib[1]; fib[5]; fib[2:5]; fib[8:4]; fib[c(2, 4, 8, 1)]
```

```
[1] 0
[1] 3
[1] 1 1 2 3
[1] 13  8  5  3  2
[1]  1  2 13  0
```

```
# Note that 'minus' means 'exclude'; no mixing of positive/negative integers!
fib[-3]; fib[-(5:9)]
```

```
 [1]   0   1   2   3   5   8  13  21  34  55  89 144
[1]   0   1   1   2  34  55  89 144
```

- selecting vector elements by logical vectors: the recycling rule is in effect

```
# selecting elements that < 10
fib[fib < 10]
```

```
[1] 0 1 1 2 3 5 8
```

```
# selecting elements that are multiples of 2 and 3
fib[fib %% 2 == 0 | fib %% 3 == 0]
```

```
[1]   0   2   3   8  21  34 144
```

```
# selecting the odd terms: 1st, 3rd, 5th, etc.
fib[c(T, F)]
```

```
[1]   0   1   3   8  21  55 144
```

- selecting vector elements by combinations of functions of logical vectors

```
v <- c(2, 8, 1, 7, -11, 6, 20, 3, 12, 35, 21, 3, 5, 7, -2);
# select all elements greater than the median
v[v > median(v)]
```

```
[1]  8  7 20 12 35 21  7
```

```
# select all elements in the lower and upper 5%
v[v < quantile(v, 0.05) | v > quantile(v, 0.95)]
```

```
[1] -11  35
```

```
# select all elements that exceed ±1 standard deviations from the mean
v[abs(v - mean(v)) > sd(v)]
```

```
[1] -11  20  35  21
```

```
# select all elements that are neither NA nor NULL
v <- c(21, 2, 3, NA, 5, NULL, NA, 32, 55)
v[!is.na(v) & !is.null(v)]
```

```
[1] 21  2  3  5 32 55
```

- generate 1-spaced number sequences — operator `:`

```
1:5; 0.5:4.5; 4.5:0.5
```

```
[1] 1 2 3 4 5
[1] 0.5 1.5 2.5 3.5 4.5
[1] 4.5 3.5 2.5 1.5 0.5
```

- generate evenly-spaced sequences: `seq`

```
seq(10, 20)
 [1] 10 11 12 13 14 15 16 17 18 19 20
seq(from = 10, to = 20, by = 1.5)
[1] 10.0 11.5 13.0 14.5 16.0 17.5 19.0
seq(from = 10, to = 20, length.out = 6)
[1] 10 12 14 16 18 20
```

- generate repetitive sequences: `rep`

```
rep('A', 5); rep(1:5, c(2, 3, 2, 1, 5))
[1] "A" "A" "A" "A" "A"
 [1] 1 1 2 2 2 3 3 4 5 5 5 5 5
```

- generate random sequences (sampling): `sample`

```
sample(LETTERS[1:5], 15, replace = T); sample(letters[1:20], 15)
 [1] "B" "E" "B" "E" "E" "E" "A" "B" "C" "B" "C" "B" "B" "D" "B"
 [1] "f" "a" "d" "s" "k" "c" "h" "l" "j" "i" "q" "n" "m" "g" "r"
```

- primitive sequences generator: sequence

```
cat(sequence(8:5, from = c(1, 2, 1, 3)))
1 2 3 4 5 6 7 8 2 3 4 5 6 7 8 1 2 3 4 5 6 3 4 5 6 7
cat(sequence(8:5, from = c(1, 2, 1)))
1 2 3 4 5 6 7 8 2 3 4 5 6 7 8 1 2 3 4 5 6 1 2 3 4 5
sequence(c(6, 7, 4), from = c(8, 1, 9), by = c(-1, 1, -2))
 [1] 8 7 6 5 4 3 1 2 3 4 5 6 7 9 7 5 3
```

- print concatenated results on screen: cat

```
# `cat` puts a space between each item: using '\n' to terminate the line
v <- c(20, 30, 3 * pi); cat("The content of v is:", v, "...\n")
The content of v is: 20 30 9.424778 ...
```

- (old school) string formatting: sprintf

```
# %3d: 3 digits; %.6f: floats with 6 decimal places; %s: string
for (i in 50 * (0:2)) {
  cat(sprintf('The %3d-th term of sequence %s(%s(%3d)) is %.6f',
              i, 'exp', 'sin', i, exp(sin(i))), '\n')
}
The   0-th term of sequence exp(sin(  0)) is 1.000000
The  50-th term of sequence exp(sin( 50)) is 0.769223
The 100-th term of sequence exp(sin(100)) is 0.602682
```

- generate concatenate string vectors: `paste`, `paste0`

```
(labels <- paste(c('a', 'b'), 1:20, sep = '-'))
```

```
 [1] "a-1"  "b-2"  "a-3"  "b-4"  "a-5"  "b-6"  "a-7"  "b-8"  "a-9"  "b-10"
[11] "a-11" "b-12" "a-13" "b-14" "a-15" "b-16" "a-17" "b-18" "a-19" "b-20"
```

```
(nth <- paste0(1:12, c("st", "nd", "rd", rep("th", 9))))
```

```
 [1] "1st"  "2nd"  "3rd"  "4th"  "5th"  "6th"  "7th"  "8th"  "9th"  "10th"
[11] "11th" "12th"
```

- appending data to a vector

```
# use c() to concatenate vector
v <- c(1, 2, 3); it <- c(6, 7, 8); c(v, it)
# R will automatically extend the vector
v <- c(1, 2, 3); v[length(v) + 1] <- 42; v
# append an entire vector
w <- c(5, 6, 7, 8); v <- c(v, w); v
v[15] <- 0; v
```

- inserting data to a vector

```
append(1:10, 99, after = 5)
append(1:10, 99, after = 0)
```

- creating a list

```
(l <- list(3.14, 'Moe', c(1, 3, 5, 2), mean))
l <- list(); l[[1]] <- 3.14; l[[2]] <- 'Moe';
l[[3]] <- c(1, 3, 5, 2); l[[4]] <- mean; l
(l <- list(max = 5, right = 0.6, mean = 20))
```

- selecting list elements

```
l[[1]]  # by position
l[['max']]; l$mean; l['max'] # by name; different outcomes
```

- building a name / value association list

```
values <- c(0.1, 0.52, 0.33); names <- c("left", "right", "mid")
l <- list(); l[names] <- values; l
# Note two conventional ways
l <- list(); l$left <- 0.1; l$right <- 0.52; l$mid <- 0.33; l
l <- list(); l[['left']] <- 0.1; l[['right']] <- 0.52; l[['mid']] <- 0.33; l
```

- removing an element from a list

```
(l <- list(date = '2020-01-05', time = '12:50:11', number = 20))
l[c('date', 'time')] <- NULL; l  # remove two elements
```

- flatten a list into a vector

```
iq.scores <- list(100, 120, 140, 112, 105);
mean(iq.scores); mean(unlist(iq.scores))
cat(iq.scores, '\n'); cat('IQ scores:', unlist(iq.scores), '\n')
```

- initializing a matrix

```
v <- 1:6; matrix(v, 2, 3)
```

- giving descriptive names to the rows and columns of a matrix

```
m <- matrix(c(1, 0.556, 0.380, 0.556, 1, 0.444, 0.380, 0.444, 1), 3, 3)
colnames(m) <- c('AAPL', 'MSFT', 'GOOG');
rownames(m) <- c('AAPL', 'MSFT', 'GOOG');
m; m['MSFT', 'GOOG']
```

- selecting one row or column from a matrix

```
m[1,]; m[1,, drop=FALSE]; m[,3]; m[, 3, drop=FALSE]
```

- initializing a data frame from column data

```
pred1 <- c(4.01, 2.64, 6.03, 2.78); pred2 <- c(10.7, 12.2, 12.2, 15.0);
pred3 <- c('AM', 'PM', 'PM', 'AM'); resp <- c(11.5, 10.0, 9.2, 5.1);
data.frame(pred1, pred2, pred3, resp)
data.frame(p1 = pred1, p2 = pred2, p3 = pred3, r = resp)
list.of.vectors <- list(p1 = pred1, p2 = pred2, p3 = pred3, r = resp)
as.data.frame(list.of.vectors)
```

- initializing a data frame from row data

```
r1 <- data.frame(a = 1, b = 2, c = "X");
r2 <- data.frame(a = 3, b = 4, c = "Y");
r3 <- data.frame(a = 5, b = 6, c = "Z");
r <- rbind(r1, r2, r3)
```

- appending rows to a data frame

```
r0 <- rbind(r, data.frame(a = 4, b = 10, c = 'u'),
               data.frame(a = 3, b = -1, c = 'xx'))
```

- removing NAs from a data frame

```
df <- data.frame(x = c(1, NA, 3, 4, 5), y = c(1, 2, NA, 4, 5))
cumsum(na.omit(df))
```

- combining two data frames

```
df1 <- data.frame(a = c(1,2)); df2 <- data.frame(b = c(7,8)); cbind(df1, df2)
df1 <- data.frame(x = c("a", "a"), y = c(5, 6))
df2 <- data.frame(x = c("b", "b"), y = c(9, 10))
rbind(df1, df2)
```

Table 1: From One Data Structure to Another: HowTo

From	To	How	Note
vector	list	`as.list(v)`	1
vector	matrix	`cbind(v)`, `as.matrix(v)`, `rbind(v)`, `matrix(v, n, m)`	
vector	dataframe	`as.data.frame(v)`, `as.data.frame(rbind(v))`	
list	vector	`unlist(v)`	2
list	matrix	`as.matrix(l)`, `as.matrix(rbind(l))`, `matrix(l, n, m)`	
list	dataframe	`as.data.frame(l)`	
matrix	vector	`as.vector(m)`	
matrix	list	`as.list(m)`	
matrix	dataframe	`as.data.frame(m)`	
dataframe	vector	`df[1,]` or `df[,1]`, `df[[1]]`	3
dataframe	list	`as.list(df)`	4
dataframe	matrix	`as.matrix(df)`	5

1. Don't use `list(v)` !
2. Use `unlist` rather than `as.vector`.
3. This makes sense only if `df` contains one row or one column.
4. Using `as.list` removes the `class` attribute `data.frame`.
5. Note the coercion hierarchy: all elements will be coerced into the common denominator type!

Programming

- format of a function:

```
function_name <- function(argument1, argument2 = default_value2, ...) {
    # ...
    # rows with code to create some_object
    # ...
    return(some_object)
}
```

- functions can access global variables

```
y_squared <- function() {
    return(y^2)
}
y <- 2; y_squared()
# [1] 4
```

- but operations on global variables inside functions won't affect global variables

```
add_to_y <- function(n) {
    y <- y + n
}
y <- 1; add_to_y(2); y
# [1] 1
```

- use `<<-` operator if you really need to change a global variable inside functions

```
add_to_y_global <- function(n) {
    y <<- y + n
}
y <- 1; add_to_y_global(2); y
# [1] 3
```

- Define function `avg(x)` to compute the average of `x`

```r
avg <- function(x) {
    return(sum(x) / length(x))
}
```

- See what happens for various inputs:

```r
avg(c(2, 3, 6)); avg(c(TRUE, FALSE, TRUE, FALSE))
```

```
# [1] 3.666667
# [1] 0.5
```

```r
avg(c('Moon', 'River', 'Wider', 'Than', 'A', 'MILE'))
```

```
# Error in sum(x): invalid 'type' (character) of argument
```

```r
avg(data.frame(x = c(5, 2, 5), y = c(-3, -1, 13)))
```

```
# [1] 10.5
```

```r
avg(data.frame(x = c(5, 2, 5), y = c('A', 'D', 'E')))
```

```
# Error in FUN(X[[i]], ...): only defined on a data frame with all numeric-alike variables
```

- Consequences of implicit `return`

```r
avg_bad <- function(x) {
    avg <- sum(x) / length(x)
}
avg_ok <- function(x) {
    sum(x) / length(x)
}
avg_bad(1:10)  # it's mute!
avg_ok(1:10)
```

```
# [1] 5.5
```

- Define function `power_n(x, n)` to compute `x` to the power of `n`

```
power_n <- function(x, n = 2){
    return(x^n)
}
```

- See what happens for various inputs:

```
power_n(3)    # using default  n = 2
# [1] 9
power_n(3, 3) # using supplied n = 3
# [1] 27
power_n(x = 5, n = 2)  # using named arguments
# [1] 25
power_n(n = 2, x = 5)  # ... even in wrong order
# [1] 25
power_n(n = 2)         # this is an error!
# Error in power_n(n = 2): argument "x" is missing, with no default
```

- It is possible to apply function name as an argument

```
apply_to_first2 <- function(x, func) {
    result <- func(x[1:2])
    return(result)
}
```

- ```r
  x <- c(1, 2, 3)
  apply_to_first2(x, sqrt)
  # [1] 1.000000 1.414214
  apply_to_first2(x, is.character)
  # [1] FALSE
  apply_to_first2(x, power_n)
  # [1] 1 4
  ```
- But what if we want to supply additional arguments to `func`?

```r
x <- c(1, 2, 3); apply_to_first2(x, sum)
# [1] 3
x <- c(NA, 2, 3); apply_to_first2(x, sum) # using sum(x, na.rm = TRUE) ...
# [1] NA
```

- `...` comes to the rescue

```r
apply_to_first2 <- function(x, func, ...) {
    result <- func(x[1:2], ...)
    return(result)
}
apply_to_first2(x, sum, na.rm = TRUE)
# [1] 2
```
```

- An example of typical usage of `if` & `else`

```r
reciprocal <- function(x) {
  if (is.na(x)) {
      cat("Error! Division by NA.")
  } else if (is.infinite(x)) {
      cat("Error! Division by infinity.")
  } else if (x == 0) {
      cat("Error! Division by zero.")
  } else {
      1 / x
  }
}
reciprocal(2); reciprocal(0); reciprocal(-Inf); reciprocal(NA)
```

```
# [1] 0.5
# Error! Division by zero.
# Error! Division by infinity.
# Error! Division by NA.
```

- `ifelse`: the vector version of `if` & `else`

```r
x <- c(-1, 2, -100, 20, 50)
ifelse(x > 0, 'positive', 'negative')
```

```
# [1] "negative" "positive" "negative" "positive" "positive"
```

- `&&` is the short-circuited `&`. Note the difference:

```r
# aa is a variable that doesn't exist; using && works:
if (exists("aa") && aa > 0) {
    cat("The variable aa exists and is positive.")
} else {
    cat("aa doesn't exist or is negative.")
}
```

```
# aa doesn't exist or is negative.
```

```r
# But using & doesn't, because it attempts to evaluate aa > 0
#     even though aa doesn't exist!
if (exists("aa") & aa > 0) {
    cat("The variable aa exists and is positive.")
} else {
    cat("aa doesn't exist or is negative.")
}
```

```
# Error in eval(expr, envir, enclos): object 'aa' not found
```

References

- Arnold, J. B. (2020). R for Data Science: Exercises Solutions. https://jrnold.github.io/r4ds-exercise-solutions/.
- Braun, J. and Murdoch, D. (2021). *A First Course in Statistical Programming with R*. Cambridge University Press, Cambridge, 3rd edition.
- Grolemund, G. and Wickham, H. (2017). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. Sebastopol, CA, O'Reilly Media, Inc. https://r4ds.had.co.nz/.
- Grosser, M., Bumann, H., and Wickham, H. (2022). *Advanced R Solutions*. Chapman & Hall/CRC, Boca Raton, FL. https://advanced-r-solutions.rbind.io/.
- Ismay, C. and Kim, Y. (2020). *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse*. Chapman & Hall/CRC, Boca Raton, FL. https://moderndive.netlify.app/.
- Long, J. D. and Teetor, P. (2019). *R Cookbook*. Sebastopol, CA, O'Reilly Media, Inc., 2nd edition. https://rc2e.com/.
- Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, San Francisco.
- Thulin, M. (2021). *Modern Statistics with R*. Eos Chasma Press. http://www.modernstatisticswithr.com/.
- Wickham, H. (2019). *Advanced R*. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition. https://adv-r.hadley.nz/.