1 Quick recap and some related issues

We can ask for more info about/help with a command by prefixing it with a “?”:

```r
> ?substr
> ?mean
```

We can get the current working directory as follows:

```r
> getwd()

```

Let’s list the files in this directory. To do that, we first save the full path to the wd in a variable:

```r
This set of notes is primarily based on Gries (2009).

1 Quick recap and some related issues

We can ask for more info about/help with a command by prefixing it with a “?”:

```r
> ?substr
> ?mean
```

We can get the current working directory as follows:

```r
> getwd()

```

Let’s list the files in this directory. To do that, we first save the full path to the wd in a variable:

```r
This set of notes is primarily based on Gries (2009).
> wd_path <- getwd()
> wd_path

> str(wd_path)
chr "/home/ady/Desktop/Dropbox/quant_methods_spring2014/lecture_notes/lecture2"

> files_in_wd <- dir(path = wd_path, all.files = F, full.names = F, 
+ recursive = T)
> files_in_wd
[1] "data_newline_sep.txt"  "data_space_sep.txt"
[3] "data_tab_sep.txt"      "gb4e-modified.sty"
[5] "quant_methods_lecture2.aux"   "quant_methods_lecture2.bbl"
[7] "quant_methods_lecture2.blg"  "quant_methods_lecture2-blx.bib"
[9] "quant_methods_lecture2.log" "quant_methods_lecture2.out"
[13] "quant_methods_lecture2.run.xml" "quant_methods_lecture2.tex"
[15] "quant_methods_lecture2.toc"

> str(files_in_wd)
chr [1:15] "data_newline_sep.txt" "data_space_sep.txt" ...

Note that “recursive=T” might not be supported on all platforms, and may be ignored, with a warning.

Indexing and slicing:

> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

> str(letters)
chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" ...

> letters[3]
[1] "c"

> x <- 3
> letters[x]
[1] "c"

> (y <- c(1, 3))
[1] 1 3
> letters[y]
[1] "a" "c"

> (z <- 1:3)
2 Patterned vectors

We start with the sequence-generating function “seq”:

```
> 1:21
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
> seq(1, 21)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
> seq(1, 21, by = 2)
[1] 1 3 5 7 9 11 13 15 17 19 21
> seq(1, 21, by = 3)
[1] 1 4 7 10 13 16 19
> seq(1, 200, by = 10)
[1] 1 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161
[18] 171 181 191
> seq(1, 21, length.out = 2)
[1] 1 21
> seq(1, 21, length.out = 3)
[1] 1 11 21
> seq(1, 21, length.out = 4)
> seq(1, 21, length.out = 5)
[1] 1 6 11 16 21
```

To repeat a value, e.g., 3, a certain number of times, e.g., 12, use “rep”: 
We can combine the two functions, e.g., we can repeat the pattern 2, 4, \ldots, 20 twice:

\begin{verbatim}
> rep(seq(2, 20, by = 2), 2)
[1] 2 4 6 8 10 12 14 16 18 20 2 4 6 8 10 12 14 16 18 20
\end{verbatim}

More examples:

\begin{verbatim}
> rep(c(1, 4), c(3, 2))  # repeat 1 three times and 4 twice
[1] 1 1 1 4 4

> rep(seq(2, 20, 2), rep(2, 10))  # repeat each value twice
[1] 2 2 4 4 6 6 8 8 10 10 12 12 14 14 16 16 18 18 20 20

> rep(seq(2, 20, 2), each = 2)  # also repeat each value twice
[1] 2 2 4 4 6 6 8 8 10 10 12 12 14 14 16 16 18 18 20 20
\end{verbatim}

Once we have a vector of numbers, we can get its minimum and maximum as follows:

\begin{verbatim}
> min(1:100)
[1] 1
> max(1:100)
[1] 100
\end{verbatim}

### 3 Logical operators

Testing for equality (note the vectorial nature):

\begin{verbatim}
> x <- letters[1:6]
> x
[1] "a" "b" "c" "d" "e" "f"
> x == "d"
[1] FALSE FALSE FALSE TRUE FALSE FALSE
\end{verbatim}

More examples and more comparison operators besides equality:

\begin{verbatim}
> x == "d"
[1] FALSE FALSE FALSE TRUE FALSE FALSE

> x <= "c"
[1] FALSE FALSE FALSE TRUE FALSE FALSE
\end{verbatim}
In addition to propositional operators, we have existential and universal quantification:

```r
> any(x == "d")
[1] TRUE
> any(x == "q")
[1] FALSE
> all(c("t", "t", "t") == "t")
[1] TRUE
> all(c("a", "t", "t") == "t")
[1] FALSE
```

We can convert logical vectors to numeric vectors:

```r
> x
[1] "a" "b" "c" "d" "e" "f"
> (x > "d" | x < "b")
[1] TRUE FALSE FALSE FALSE TRUE TRUE
```
```r
> as.numeric(x > "d" | x < "b")
[1] 1 0 0 0 1 1
> sum(x > "d" | x < "b")  # this converts to numeric automatically
[1] 3

Tangentially, this is how we can reverse a vector and sort it:

> z <- rev(x)
> z
[1] "f" "e" "d" "c" "b" "a"
> sort(z)
[1] "a" "b" "c" "d" "e" "f"

### 3.1 Indexing/slicing with logical operators

```
We can also do indexing/slicing by storing intermediate results in variables:

```r
> x1 <- which(x > "d" | x < "b")
> x1
[1] 1 5 6
> x2 <- x[x1]
> x2
[1] "a" "e" "f"
```

Finally, we can combine indexing/slicing and assignment of values to variables:

```r
> x3 <- which(x > "d")
> x[x3] <- "w"
> x
[1] "a" "b" "c" "d" "w" "w"
```

Or in one go:

```r
> x[x < "c"] <- "z"
> x
[1] "z" "z" "c" "d" "w" "w"
```

### 4 Set operators

```r
> (x <- c(10:1))
[1] 10 9 8 7 6 5 4 3 2 1
> (y <- c(2, 5, 9))
[1] 2 5 9
```

Set membership:

```r
> x %in% y
[1] FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE
> y %in% x
[1] TRUE TRUE TRUE
```

```r
> sum(x %in% y)
[1] 3
> sum(y %in% x)
[1] 3
> x[x %in% y]
[1] 9 5 2
```
More examples for set membership:

```r
> (x1 <- letters[c(10:1)])
[1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
> (y1 <- letters[c(2, 5, 9)])
[1] "b" "e" "i"
> x1 %in% y1
[1] FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE
> y1 %in% x1
[1] TRUE TRUE TRUE
> sum(x1 %in% y1)
[1] 3
> sum(y1 %in% x1)
[1] 3
> x1[x1 %in% y1]
[1] "i" "e" "b"
```

Matching – finer-grained, ‘vectorial’ (coordinate / index based) membership:

```r
> # ?match
> x
 [1] 10  9  8  7  6  5  4  3  2  1
> y
 [1]  2  5  9
> match(x, y)
 [1] NA  3 NA NA NA  2 NA NA  1 NA
> match(y, x)
 [1]  9  6  2
> x1
 [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
> y1
 [1] "b" "e" "i"
> match(x1, y1)
 [1] NA  3 NA NA NA  2 NA NA  1 NA
> match(y1, x1)
 [1]  9  6  2
```
Set difference:

\[ \text{setdiff}(x, y) \]
\[ 10 \ 8 \ 7 \ 6 \ 4 \ 3 \ 1 \]
\[ x[-y] \quad \# \text{not that this is NOT set difference} \]
\[ 10 \ 8 \ 7 \ 5 \ 4 \ 3 \ 1 \]
\[ \text{setdiff}(y, x) \]
numeric(0)
\[ \text{setdiff}(x_1, y_1) \]
\[ "j" "h" "g" "f" "d" "c" "a" \]
\[ x_1[-\text{which}(x_1 \ %\in\ % y_1)] \quad \# \text{but this IS set difference} \]
\[ "j" "h" "g" "f" "d" "c" "a" \]
\[ \text{setdiff}(y_1, x_1) \]
\[ \text{character}(0) \]

Set intersection:

\[ \text{intersect}(x, y) \]
\[ 9 \ 5 \ 2 \]
\[ \text{intersect}(y, x) \]
\[ 2 \ 5 \ 9 \]
\[ \text{intersect}(x_1, y_1) \]
\[ "i" "e" "b" \]
\[ \text{intersect}(y_1, x_1) \]
\[ "b" "e" "i" \]

Set union:

\[ \text{union}(x, y) \]
\[ 10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \]
\[ \text{union}(y, x) \]
\[ 2 \ 5 \ 9 \ 10 \ 8 \ 7 \ 6 \ 4 \ 3 \ 1 \]
\[ \text{union}(x_1, y_1) \]
\[ "j" "i" "h" "g" "f" "e" "d" "c" "b" "a" \]
\[ \text{union}(y_1, x_1) \]
\[ "b" "e" "i" "j" "h" "g" "f" "d" "c" "a" \]
5 Counting

Unique elements in a vector:

```r
> (f <- c(1, 2, 3, 2, 3, 4, 3, 4, 5))
[1] 1 2 3 2 3 4 3 4 5
> unique(f)
[1] 1 2 3 4 5
> (g <- c(2, 3, 1, 5, 2, 6, 3, 1, 2))
[1] 2 3 1 5 2 6 3 1 2
> unique(g)
[1] 2 3 1 5 6
> sort(unique(g))
[1] 1 2 3 5 6
```

Another example:

```r
> (f1 <- c("a", "b", "c", "b", "c", "d", "c", "d", "e"))
[1] "a" "b" "c" "b" "c" "d" "c" "d" "e"
> unique(f1)
[1] "a" "b" "c" "d" "e"
> (g1 <- c("b", "c", "a", "e", "b", "f", "c", "a", "b"))
[1] "b" "c" "a" "e" "b" "f" "c" "a" "b"
> unique(g1)
[1] "b" "c" "a" "e" "f"
> sort(unique(g1))
[1] "a" "b" "c" "e" "f"
```

Tables of counts, cross-tabulations, tables of proportions:

```r
> table(f)
 f
 1 2 3 4 5
1 2 3 1
2 1 2
> table(f1)
 f1
 a b c d e
1 2 3 2 1
```
> xtabs(~f)

f
1 2 3 4 5
1 2 3 2 1

> xtabs(~f1)

f1
a b c d e
1 2 3 2 1

> table(g1)

g1
a b c e f
2 3 2 1 1

> xtabs(~g1)

g1
a b c e f
2 3 2 1 1

> table(f)

f
1 2 3 4 5
1 2 3 2 1

> prop.table(table(f))

f

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.1111</td>
<td>0.2222</td>
<td>0.3333</td>
<td>0.2222</td>
<td>0.1111</td>
</tr>
</tbody>
</table>

> table(f1)

f1
a b c d e
1 2 3 2 1

> prop.table(table(f1))

f1

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1111</td>
<td>0.2222</td>
<td>0.3333</td>
<td>0.2222</td>
<td>0.1111</td>
</tr>
</tbody>
</table>

> f1

[1] "a" "b" "c" "b" "c" "d" "c" "d" "e"

> g1

[1] "b" "c" "a" "e" "b" "f" "c" "a" "b"

> table(f1, g1)
```r
> prop.table(table(f1, g1))

> prop.table(table(f1, g1), 1)

> prop.table(table(f1, g1), 2)

6 Sorting

> g
[1] 2 3 1 5 2 6 3 1 2
> sort(g, decreasing = T)
[1] 6 5 3 3 2 2 2 1 1
> g
[1] 2 3 1 5 2 6 3 1 2
```
> sort(g, decreasing = F)
[1] 1 1 2 2 2 3 3 5 6
> sort(g)
[1] 1 1 2 2 2 3 3 5 6
> (z <- c(3, 5, 10, 1, 6, 7, 8, 2, 4, 9))
[1] 3 5 10 1 6 7 8 2 4 9
> order(z, decreasing = F)
[1] 4 8 1 9 2 5 6 7 10 3
> z[order(z, decreasing = F)]
[1] 1 2 3 4 5 6 7 8 9 10
> sort(z, decreasing = F)
[1] 1 2 3 4 5 6 7 8 9 10
> order(z, decreasing = T)
[1] 3 10 7 6 5 2 9 1 8 4
> z[order(z, decreasing = T)]
[1] 10 9 8 7 6 5 4 3 2 1
> sort(z, decreasing = T)
[1] 10 9 8 7 6 5 4 3 2 1

7 Printing / Saving to a file

> cat(z, file = "data_newline_sep.txt", sep = "\n", append = F)
> cat(z, file = "data_space_sep.txt", sep = " ", append = T)
> cat(z, file = "data_tab_sep.txt", sep = "\t", append = T)

8 Factors

Factors are variables with discrete (usually finite) values. For example:

- grammatical categories: noun, verb, adjective etc.
- whether a grammatical category is open (new members of that category are / can be created freely) or closed (the members of that category are a more or less fixed set)
> (f <- c("open", "open", "open", "closed", "closed"))
[1] "open" "open" "open" "closed" "closed"
> (f <- factor(f))
[1] open open open closed closed
Levels: closed open
> str(f)
Factor w/ 2 levels "closed", "open": 2 2 2 1 1
> (f <- c("open", "open", "open", "closed", "closed"))
[1] "open" "open" "open" "closed" "closed"
> (f <- as.factor(f))
[1] open open open closed closed
Levels: closed open
> str(f)
Factor w/ 2 levels "closed", "open": 2 2 2 1 1
> levels(f)
[1] "closed" "open"
> levels(f) <- c("blah1", "blah2")
> str(f)
Factor w/ 2 levels "blah1", "blah2": 2 2 2 1 1
> f
[1] blah2 blah2 blah2 blah1 blah1
Levels: blah1 blah2
> (f <- as.integer(f))
[1] 2 2 2 1 1

References


