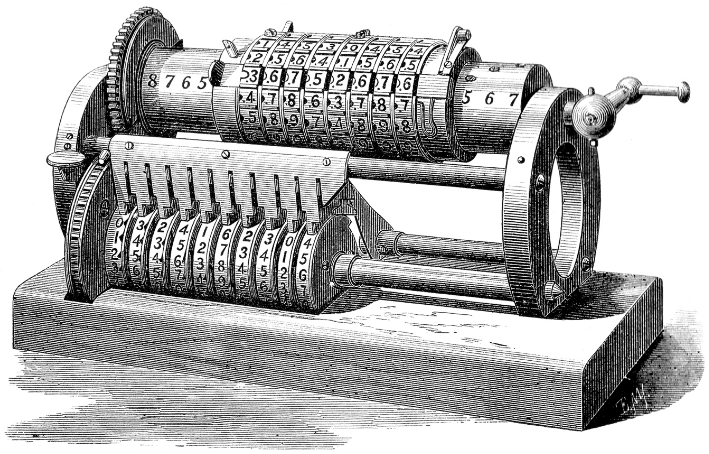
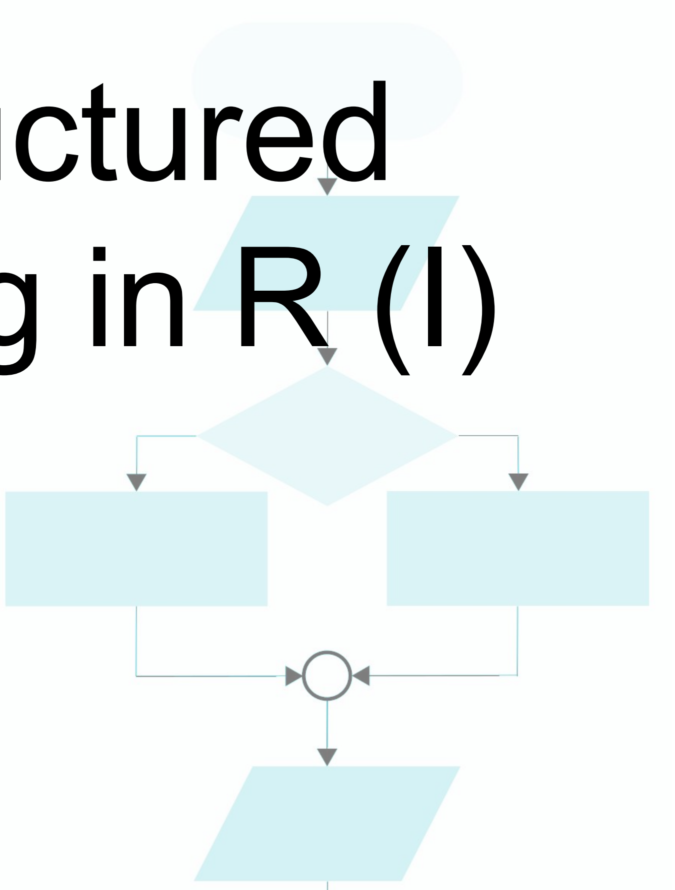
Programming in R

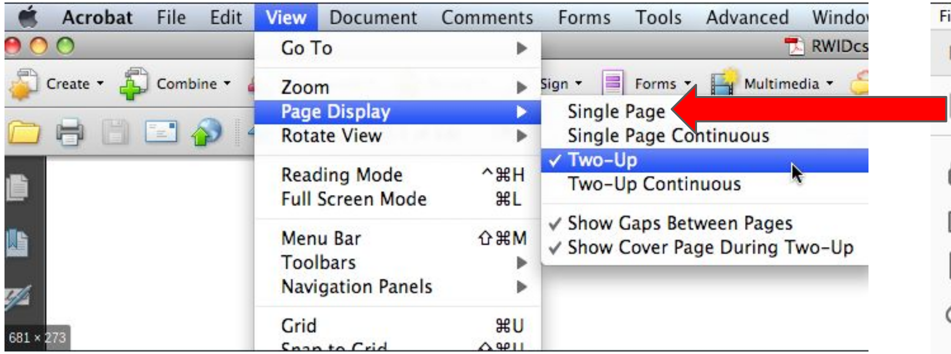


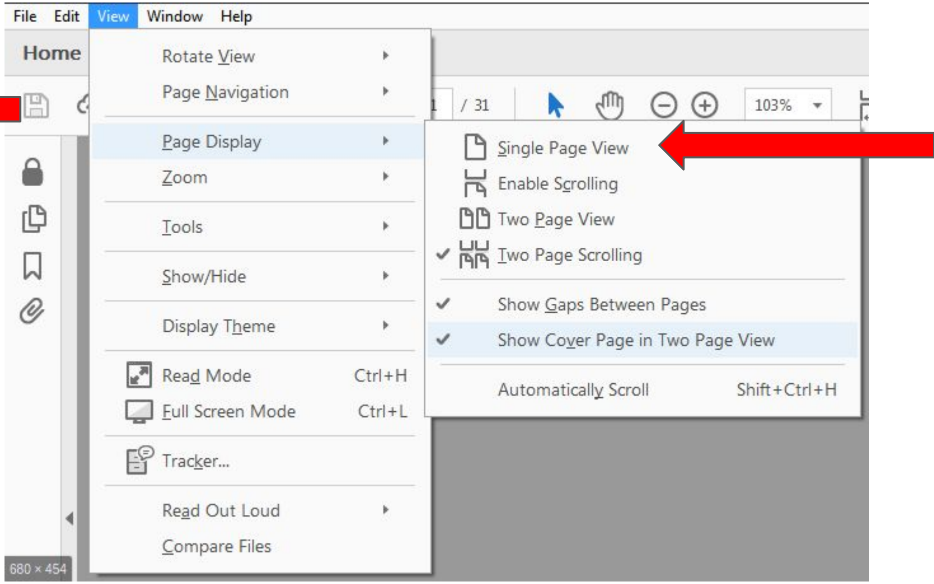
Unit 2: Structured Programming in R (I)



About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Structured Programming in R

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "R" Track: Data Types and Control Structures
- "Dev" Track: Programming Style

R Track

Data Types and Control Structures

"Structured Programming"

- Using conditionals ("if..then"), loops ("for"), and functions to solve a problem
- ... as opposed to, say, copy-pasting your code to execute the same action multiple times[1]
  - people actually do that and it makes for entertaining stories. Dont. Do. It.
- This course unit is about basic structured programming in R and covers a large part of the content of last year's course.
  - Datatypes (atomics, lists)
  - Operators
  - Control structures (conditionals, loops, functions)
- We will go a bit beyond last year to get you a solid foundational knowledge of the "nuts and bolts" in R!

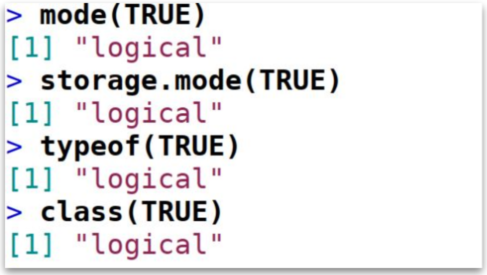
[1]In some situations there is a legitimate technique called "loop unrolling" for performance reasons. Don't do it in R; if you need performance in R you use Rcpp or similar.

Reminder: Data Types

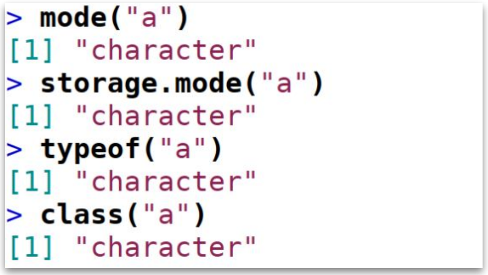
- "atomic" data types: `logical`, `numeric`, `character`
  - (also less importantly: `integer`, `complex`, `raw`)
  - "vectors"
- "recursive" data types: `list` (and some other more special things)

Reminder: Data Types

Detect types using mode(), storage.mode(), typeof(), class()



```
> mode(TRUE)
[1] "logical"
> storage.mode(TRUE)
[1] "logical"
> typeof(TRUE)
[1] "logical"
> class(TRUE)
[1] "logical"
```

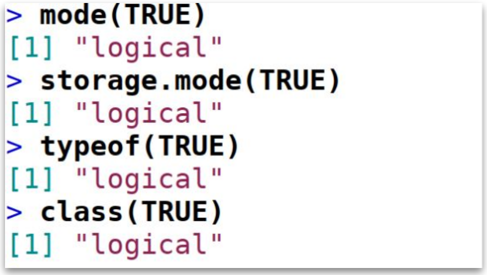


```
> mode("a")
[1] "character"
> storage.mode("a")
[1] "character"
> typeof("a")
[1] "character"
> class("a")
[1] "character"
```

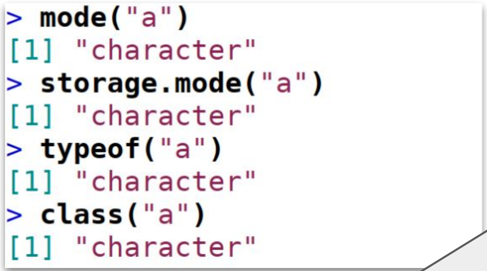
(Looks harmless at first sight)

Reminder: Data Types

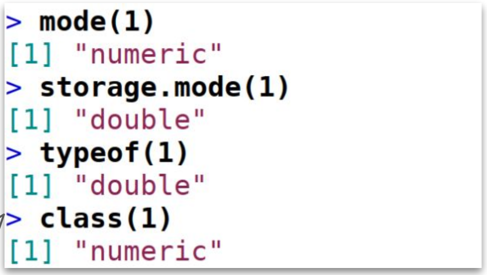
Detect types using mode(), storage.mode(), typeof(), class()



```
> mode(TRUE)
[1] "logical"
> storage.mode(TRUE)
[1] "logical"
> typeof(TRUE)
[1] "logical"
> class(TRUE)
[1] "logical"
```



```
> mode("a")
[1] "character"
> storage.mode("a")
[1] "character"
> typeof("a")
[1] "character"
> class("a")
[1] "character"
```

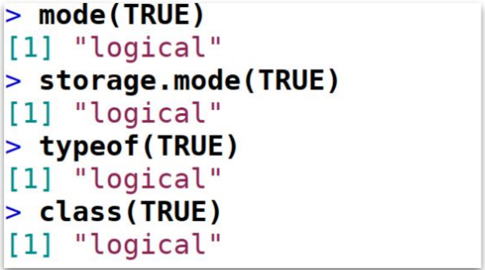


```
> mode(1)
[1] "numeric"
> storage.mode(1)
[1] "double"
> typeof(1)
[1] "double"
> class(1)
[1] "numeric"
```

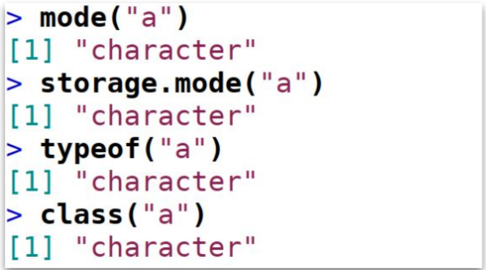
Some differences in naming here. `double` stands for "double precision", since they use 64 bits, twice as much as "single precision" numbers that people apparently considered the default at some point.

Reminder: Data Types

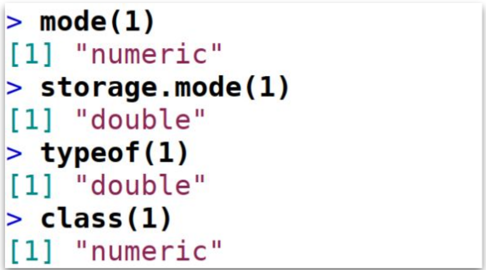
Detect types using mode(), storage.mode(), typeof(), class()



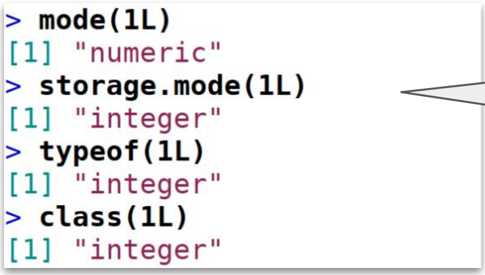
```
> mode(TRUE)
[1] "logical"
> storage.mode(TRUE)
[1] "logical"
> typeof(TRUE)
[1] "logical"
> class(TRUE)
[1] "logical"
```



```
> mode("a")
[1] "character"
> storage.mode("a")
[1] "character"
> typeof("a")
[1] "character"
> class("a")
[1] "character"
```



```
> mode(1)
[1] "numeric"
> storage.mode(1)
[1] "double"
> typeof(1)
[1] "double"
> class(1)
[1] "numeric"
```

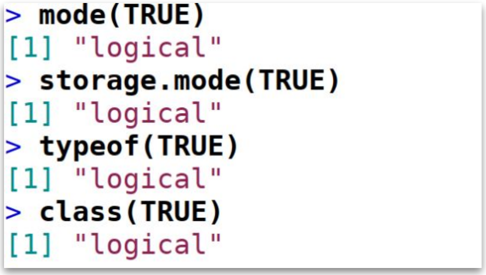


```
> mode(1L)
[1] "numeric"
> storage.mode(1L)
[1] "integer"
> typeof(1L)
[1] "integer"
> class(1L)
[1] "integer"
```

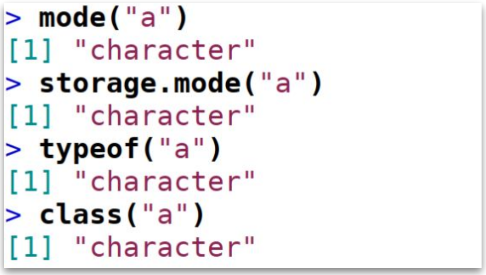
`integer` numbers are very similar to `integer` (real-valued) numbers in R.

Reminder: Data Types

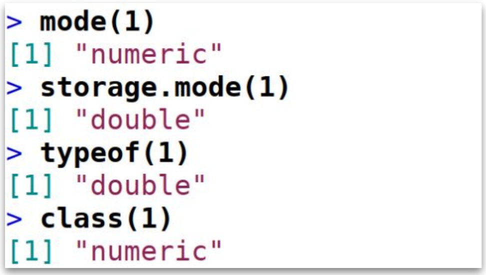
Detect types using mode(), storage.mode(), typeof(), class()



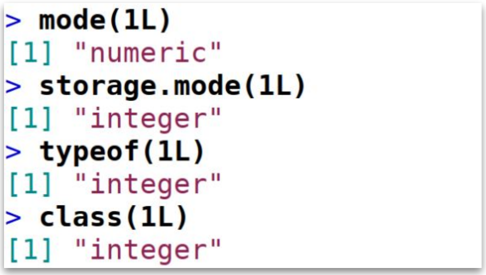
```
> mode(TRUE)
[1] "logical"
> storage.mode(TRUE)
[1] "logical"
> typeof(TRUE)
[1] "logical"
> class(TRUE)
[1] "logical"
```



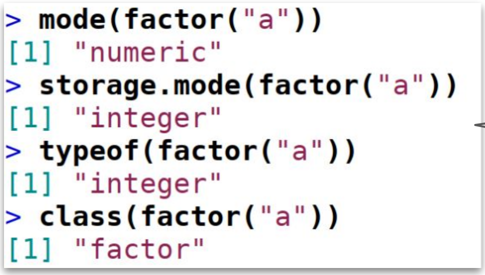
```
> mode("a")
[1] "character"
> storage.mode("a")
[1] "character"
> typeof("a")
[1] "character"
> class("a")
[1] "character"
```



```
> mode(1)
[1] "numeric"
> storage.mode(1)
[1] "double"
> typeof(1)
[1] "double"
> class(1)
[1] "numeric"
```



```
> mode(1L)
[1] "numeric"
> storage.mode(1L)
[1] "integer"
> typeof(1L)
[1] "integer"
> class(1L)
[1] "integer"
```



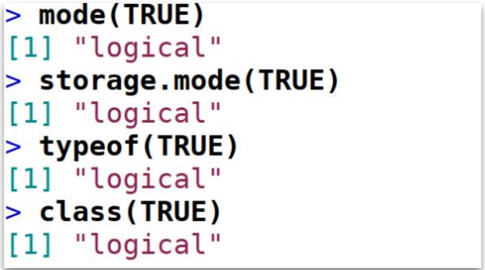
```
> mode(factor("a"))
[1] "numeric"
> storage.mode(factor("a"))
[1] "integer"
> typeof(factor("a"))
[1] "integer"
> class(factor("a"))
[1] "factor"
```

Internally, a `factor` is an integer vector with additional attributes and a different class.

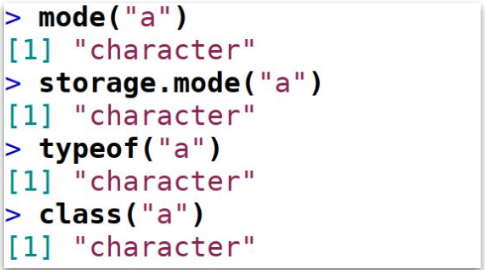
Reminder: Data Types

Detect types using mode(), storage.mode(), typeof(), class()

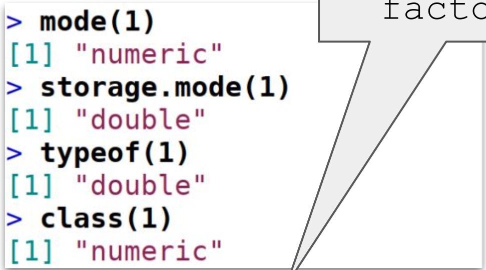
`ordered()`: mostly similar to `factor()`



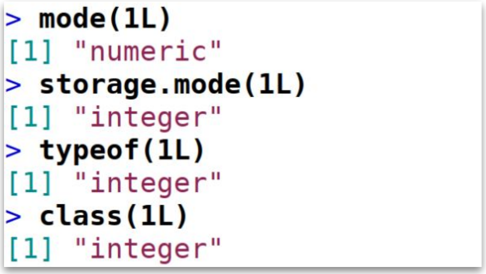
```
> mode(TRUE)
[1] "logical"
> storage.mode(TRUE)
[1] "logical"
> typeof(TRUE)
[1] "logical"
> class(TRUE)
[1] "logical"
```



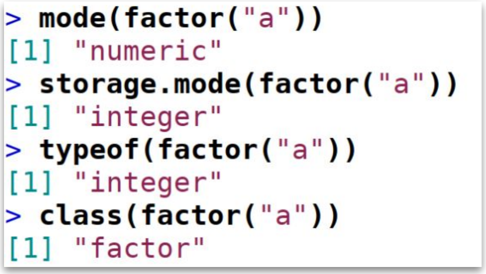
```
> mode("a")
[1] "character"
> storage.mode("a")
[1] "character"
> typeof("a")
[1] "character"
> class("a")
[1] "character"
```



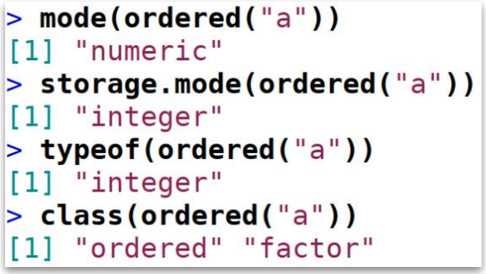
```
> mode(1)
[1] "numeric"
> storage.mode(1)
[1] "double"
> typeof(1)
[1] "double"
> class(1)
[1] "numeric"
```



```
> mode(1L)
[1] "numeric"
> storage.mode(1L)
[1] "integer"
> typeof(1L)
[1] "integer"
> class(1L)
[1] "integer"
```



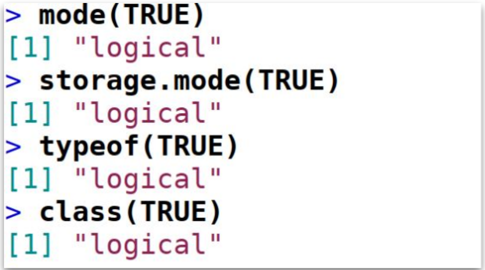
```
> mode(factor("a"))
[1] "numeric"
> storage.mode(factor("a"))
[1] "integer"
> typeof(factor("a"))
[1] "integer"
> class(factor("a"))
[1] "factor"
```



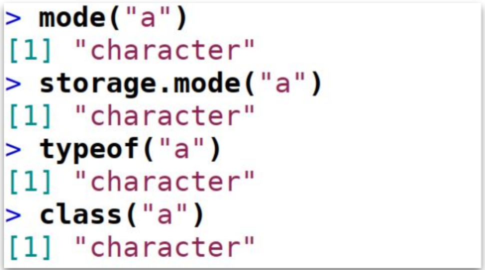
```
> mode(ordered("a"))
[1] "numeric"
> storage.mode(ordered("a"))
[1] "integer"
> typeof(ordered("a"))
[1] "integer"
> class(ordered("a"))
[1] "ordered" "factor"
```

Reminder: Data Types

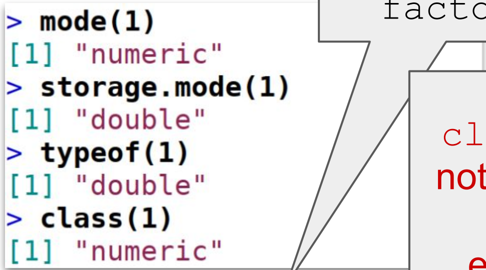
Detect types using mode(), storage.mode(), typeof(), class()



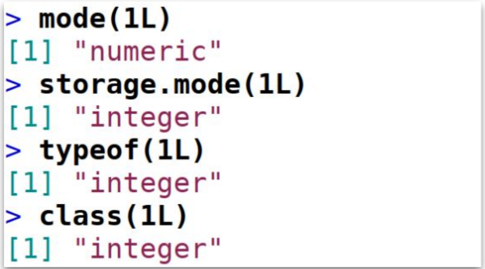
```
> mode(TRUE)
[1] "logical"
> storage.mode(TRUE)
[1] "logical"
> typeof(TRUE)
[1] "logical"
> class(TRUE)
[1] "logical"
```



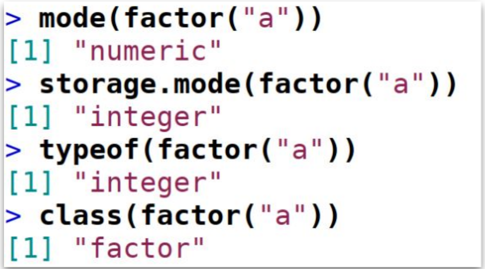
```
> mode("a")
[1] "character"
> storage.mode("a")
[1] "character"
> typeof("a")
[1] "character"
> class("a")
[1] "character"
```



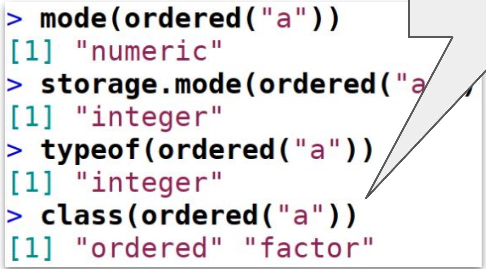
```
> mode(1)
[1] "numeric"
> storage.mode(1)
[1] "double"
> typeof(1)
[1] "double"
> class(1)
[1] "numeric"
```



```
> mode(1L)
[1] "numeric"
> storage.mode(1L)
[1] "integer"
> typeof(1L)
[1] "integer"
> class(1L)
[1] "integer"
```



```
> mode(factor("a"))
[1] "numeric"
> storage.mode(factor("a"))
[1] "integer"
> typeof(factor("a"))
[1] "integer"
> class(factor("a"))
[1] "factor"
```



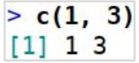
```
> mode(ordered("a"))
[1] "numeric"
> storage.mode(ordered("a
[1] "integer"
> typeof(ordered("a"))
[1] "integer"
> class(ordered("a"))
[1] "ordered" "factor"
```

`ordered()`: mostly similar to `factor()`

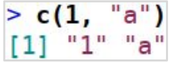
`class()` is not always a single element!

Reminder: Data Types

- **atomic** data types: `logical`, `numeric`, `character`
  - (also less importantly: `integer`, `complex`, `raw`)
  - "vectors"
- "atomic": always a vector of zero or more values *of the same type*
- no nesting, no recursive structure

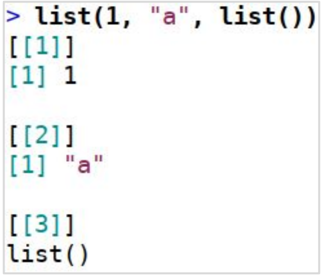


```
> c(1, 3)
[1] 1 3
```



```
> c(1, "a")
[1] "1" "a"
```

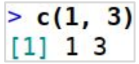
- **recursive** data types: `list`
  - (and some other things we omit for now)
- may contain any other values



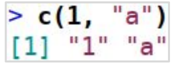
```
> list(1, "a", list())
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
list()
```

Reminder: Data Types

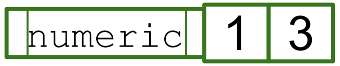
- **atomic** data types: `logical`, `numeric`, `character`
  - (also less importantly: `integer`, `complex`, `raw`)
  - "vectors"
- "atomic": always a vector of zero or more values *of the same type*
- no nesting, no recursive structure

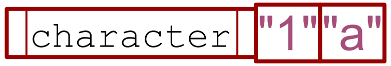


```
> c(1, 3)
[1] 1 3
```

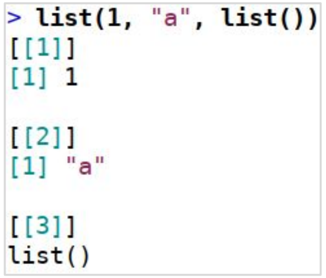


```
> c(1, "a")
[1] "1"  "a"
```

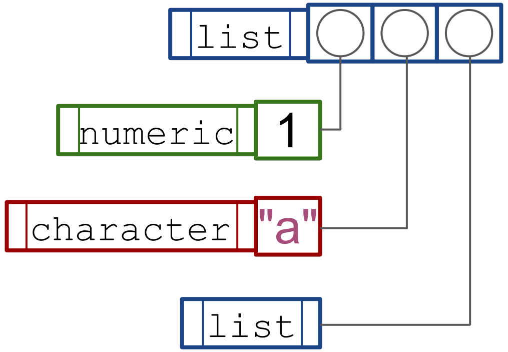




- **recursive** data types: `list`
  - (and some other things we omit for now)
- may contain any other values

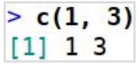


```
> list(1, "a", list())
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
list()
```

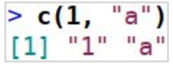


Reminder: Data Types

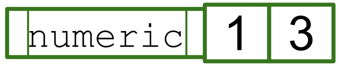
- **atomic** data types: `logical`, `numeric`, `character`
  - (also less importantly: `integer`, `complex`, `raw`)
  - "vectors"
- "atomic": always a vector of zero or more values *of the same type*
- no nesting, no recursive structure

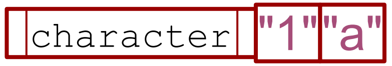


```
> c(1, 3)
[1] 1 3
```

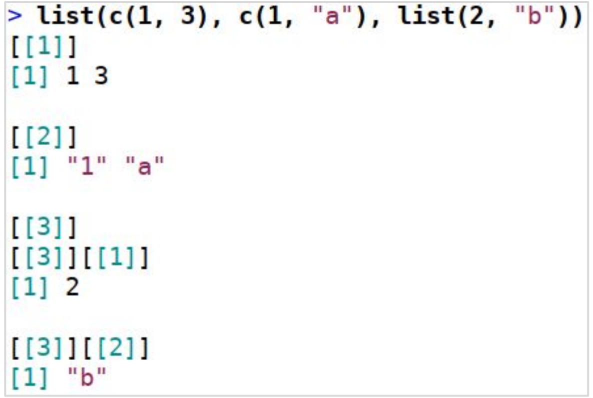


```
> c(1, "a")
[1] "1" "a"
```





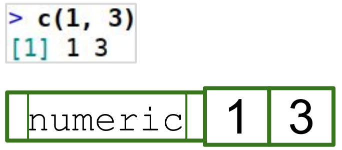
- **recursive** data types: `list`
  - (and some other things we omit for now)
- may contain any other values



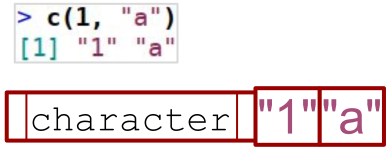
```
> list(c(1, 3), c(1, "a"), list(2, "b"))
[[1]]
[1] 1 3

[[2]]
[1] "1" "a"

[[3]]
[[3]][[1]]
[1] 2

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

Reminder: Data Types

- **atomic** data types: `logical`, `numeric`, `character`
  - (also less importantly: `integer`, `complex`, `raw`)
  - "vectors"
- "atomic": always a vector of zero or more values *of the same type*
- no nesting, no recursive structure

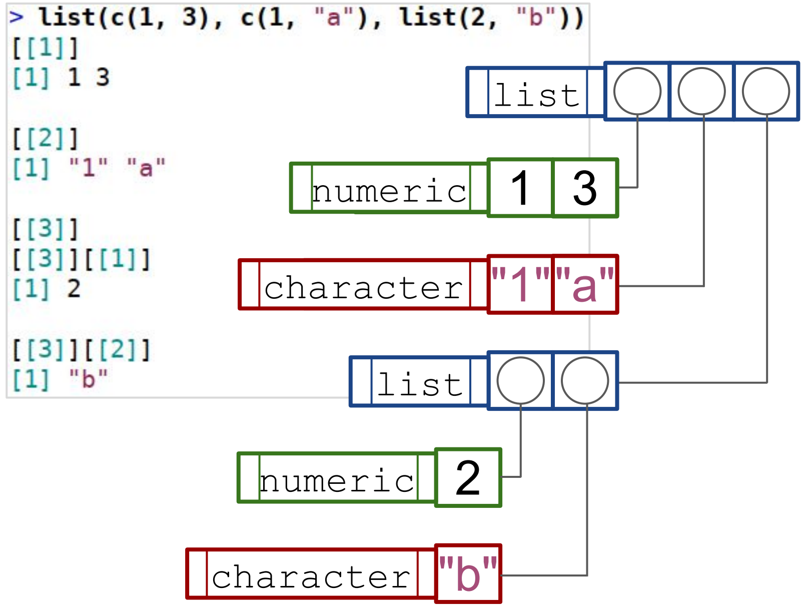


```
> c(1, 3)
[1] 1 3
```



```
> c(1, "a")
[1] "1" "a"
```

- **recursive** data types: `list`
  - (and some other things we omit for now)
- may contain any other values

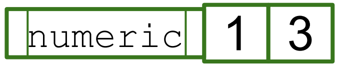


```
> list(c(1, 3), c(1, "a"), list(2, "b"))
[[1]]
[1] 1 3

[[2]]
[1] "1" "a"

[[3]]
[[3]][[1]]
[1] 2

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

Reminder: Data Types

- atomic types: indexing

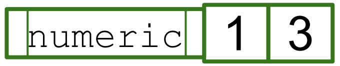
```
> x <- c(1, 3)
```



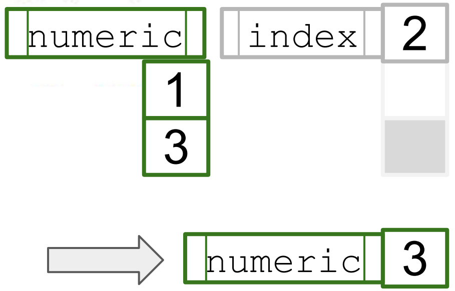
Reminder: Data Types

- atomic types: indexing

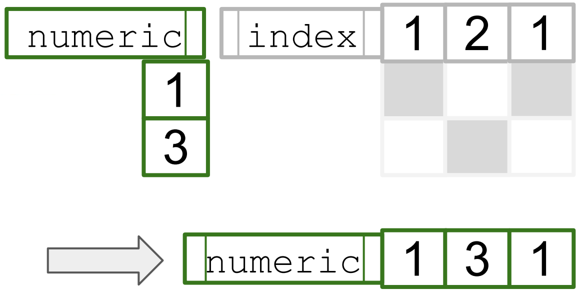
```
> x <- c(1, 3)
```



```
> x[2]
[1] 3
```



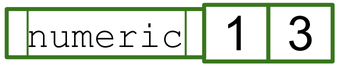
```
> x[c(1, 2, 1)]
[1] 1 3 1
```



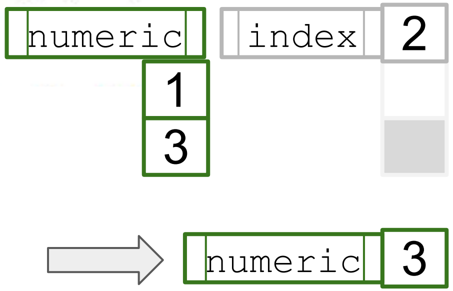
Reminder: Data Types

- atomic types: indexing

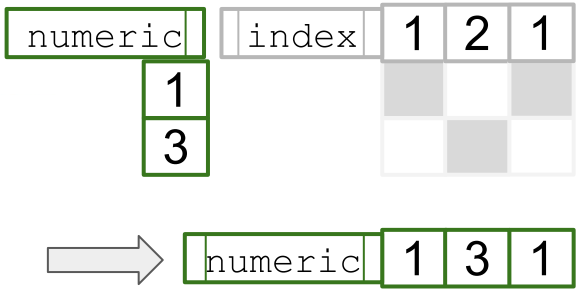
```
> x <- c(1, 3)
```



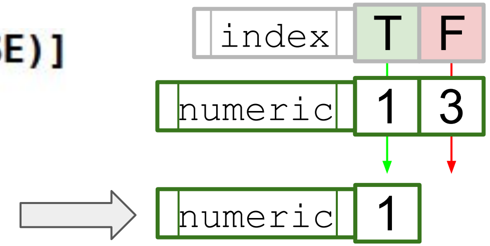
```
> x[2]
[1] 3
```



```
> x[c(1, 2, 1)]
[1] 1 3 1
```



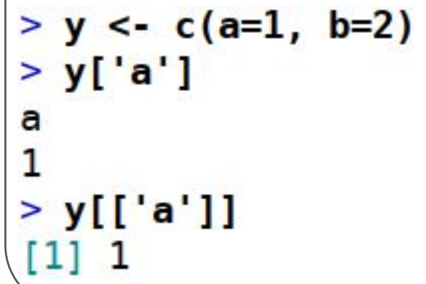
```
> x[c(TRUE, FALSE)]
[1] 1
```



Reminder: Data Types

- atomic types: indexing
  - single brackets [ ]: index with integers (or character names) or logicals
  - double brackets [[ ]]: index with single integer (or single character name) only get single entry. **This is preferred if you know you just need a single entry.**

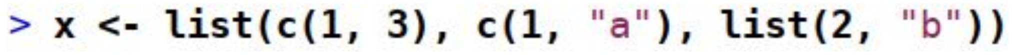
Named indexing of vectors:



```
> y <- c(a=1, b=2)
> y['a']
a
1
> y[['a']]
[1] 1
```

Reminder: Data Types

- list indexing

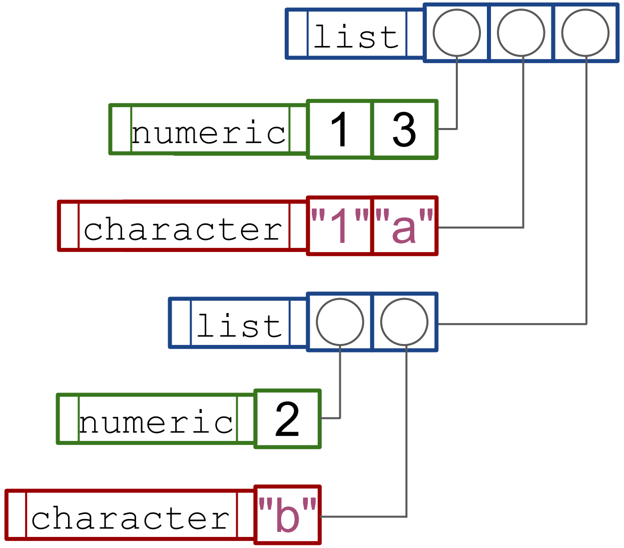


```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

Reminder: Data Types

- list indexing

```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

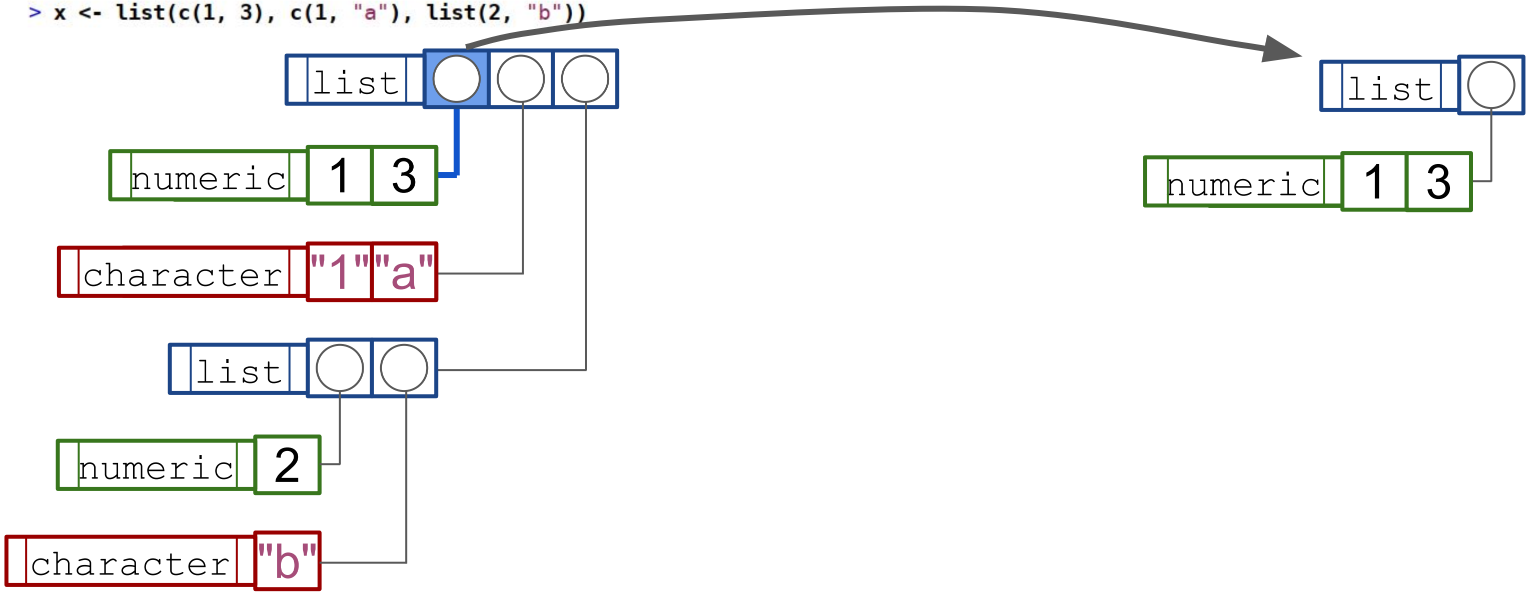


Reminder: Data Types

```
> x[1]
[[1]]
[1] 1 3
```

- list indexing: [ ] just gives a smaller list

```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

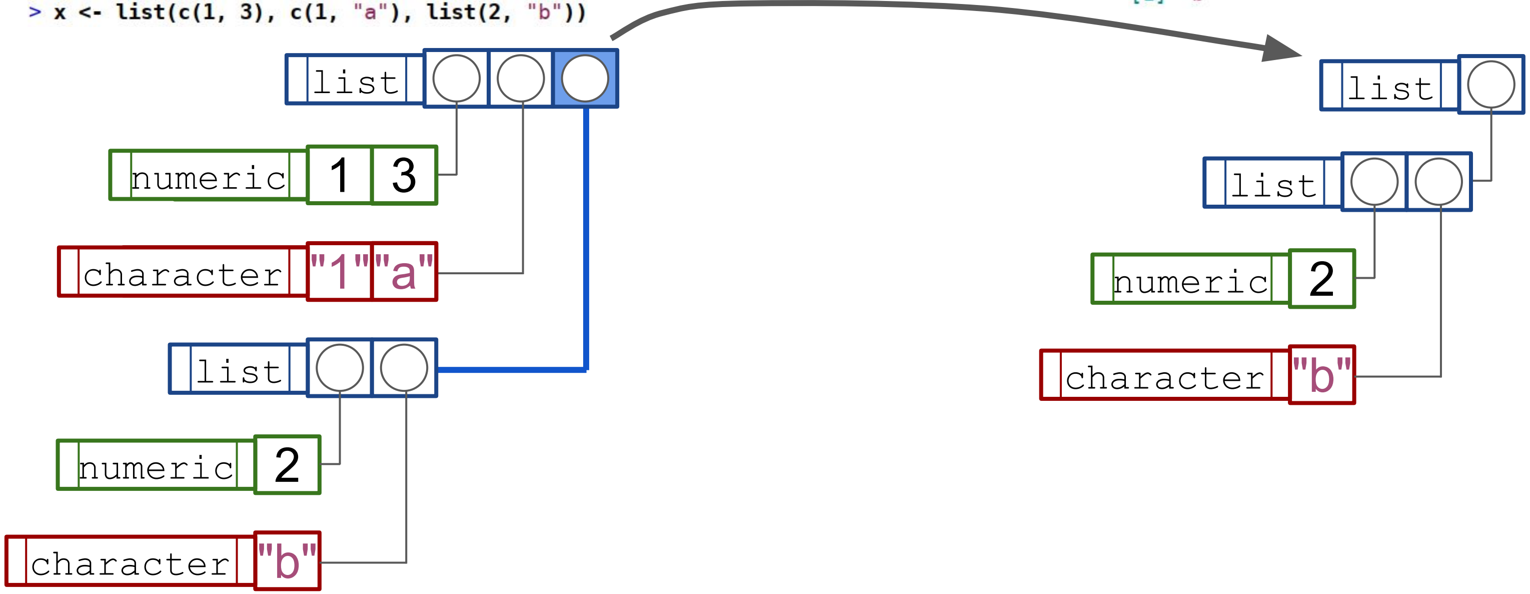


Reminder: Data Types

- list indexing: [ ] just gives a smaller list

```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

```
> x[3]
[[1]]
[[1]][[1]]
[1] 2

[[1]][[2]]
[1] "b"
```

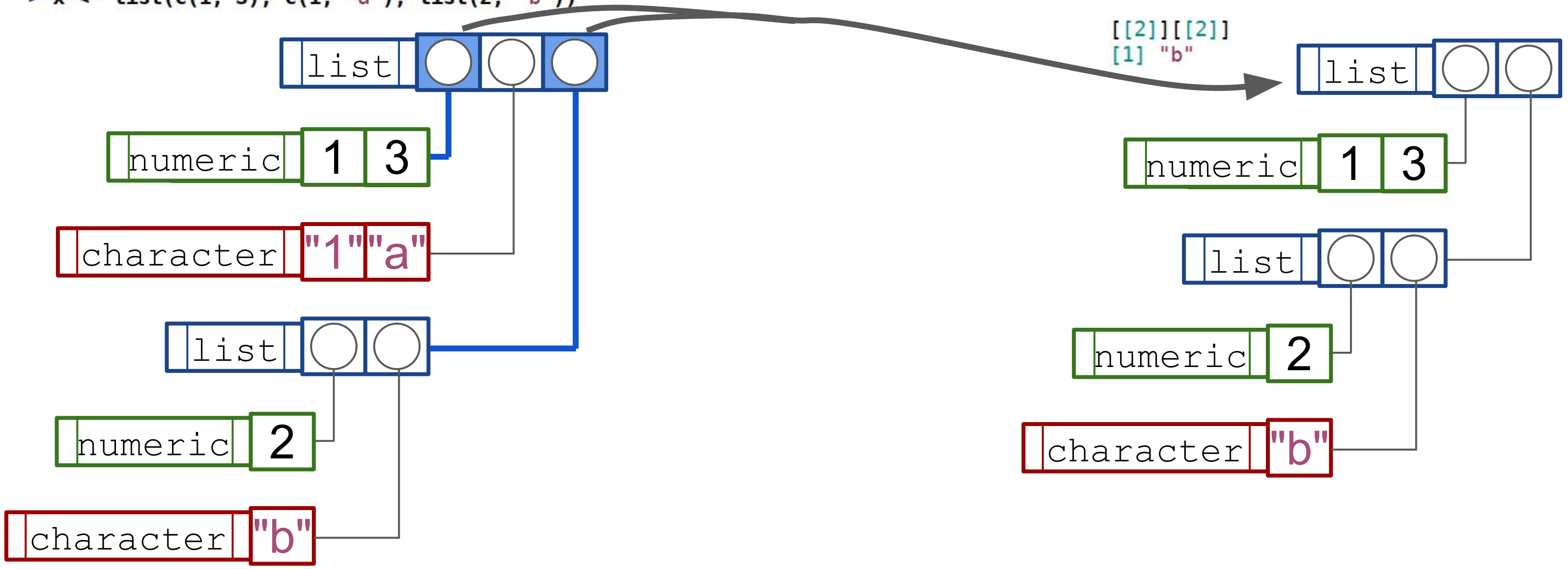


Reminder: Data Types

- list indexing: [ ] just gives a smaller list

```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

```
> x[c(1, 3)]
[[1]]
[1] 1 3

[[2]]
[[2]][[1]]
[1] 2

[[2]][[2]]
[1] "b"
```

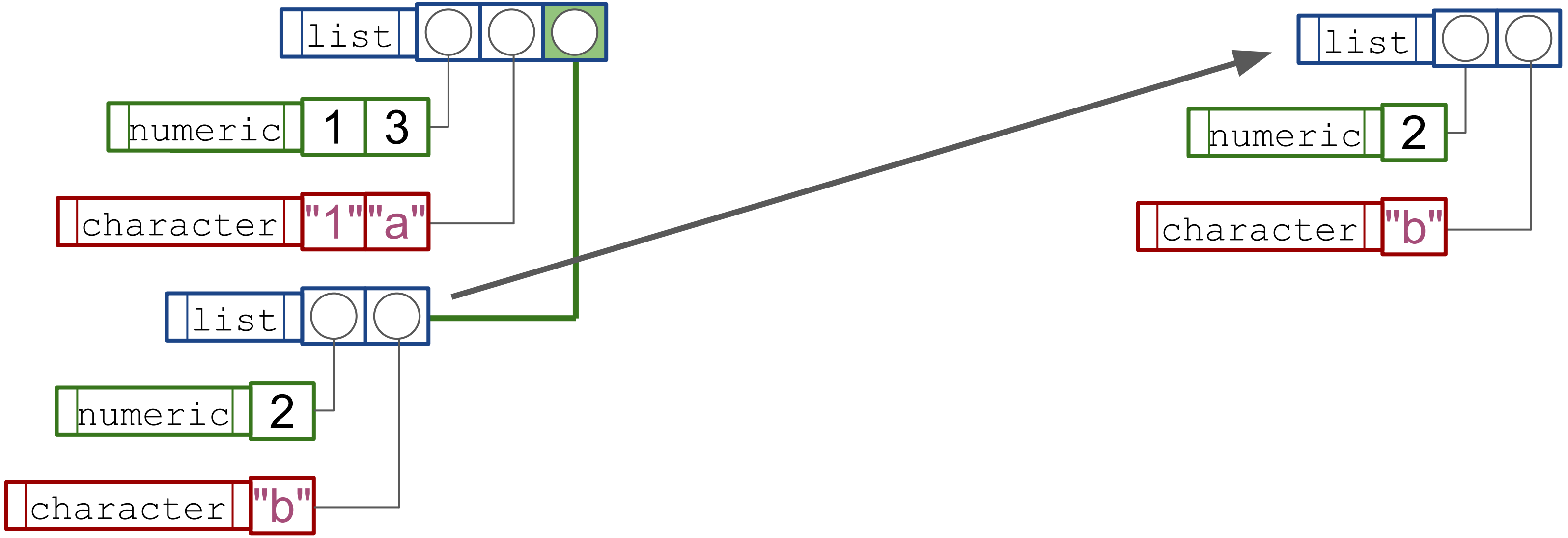


Reminder: Data Types

- list indexing: [[ ]] gives *content* of list element

```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

```
> x[[3]]
[[1]]
[1] 2

[[2]]
[1] "b"
```

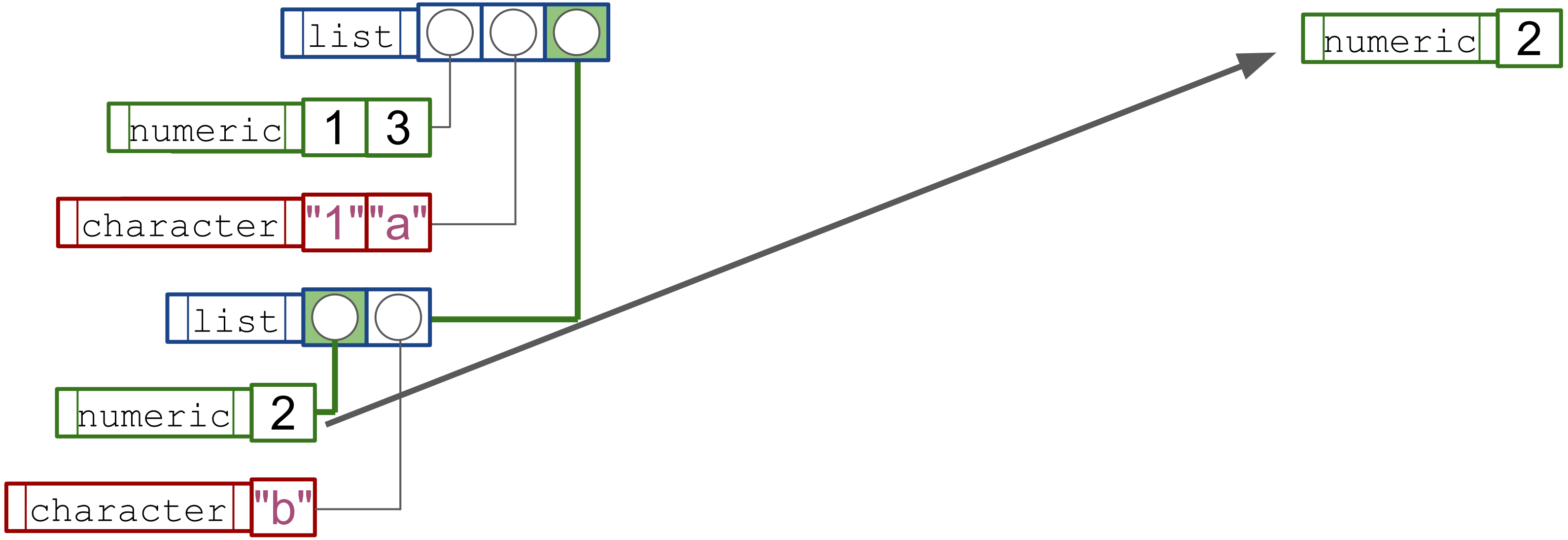


Reminder: Data Types

```
> x[[3]][[1]]
[1] 2
```

- list indexing: [[ ]] gives *content* of list element

```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```

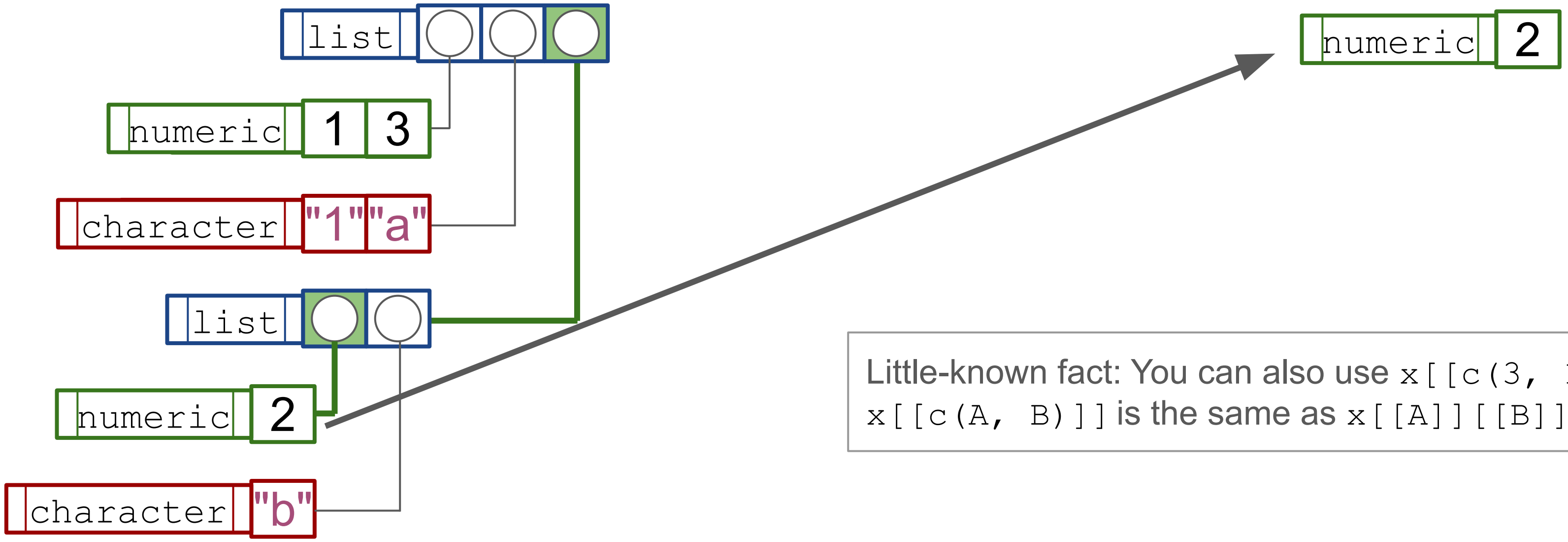


Reminder: Data Types

```
> x[[3]][[1]]
[1] 2
```

- list indexing: [[ ]] gives *content* of list element

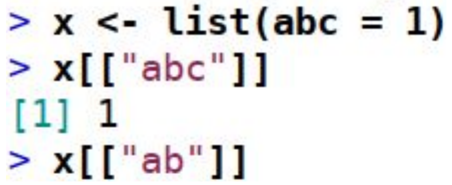
```
> x <- list(c(1, 3), c(1, "a"), list(2, "b"))
```



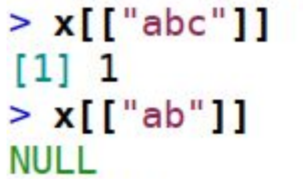
Little-known fact: You can also use `x[[c(3, 1)]]` here: `x[[c(A, B)]]` is the same as `x[[A]][[B]]`.

Reminder: Data Types

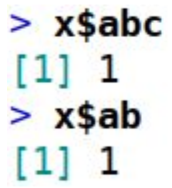
- list indexing:
  - [ ] just gives a smaller list
  - [[ ]] gives *content* of list element
  - *both* of these work with names as well
  - the `$`-operator is also common, but note the difference to [[ ]]:



```
> x <- list(abc = 1)
> x[["abc"]]
[1] 1
> x[["ab"]]
```



```
> x[["abc"]]
[1] 1
> x[["ab"]]
NULL
```

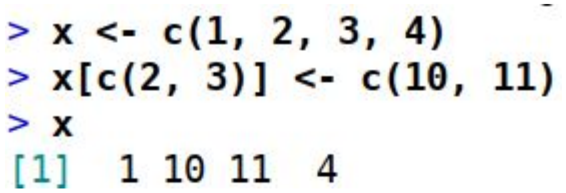


```
> x$abc
[1] 1
> x$ab
[1] 1
```

This is "partial matching", avoid this if possible!

Reminder: Subset Assignment

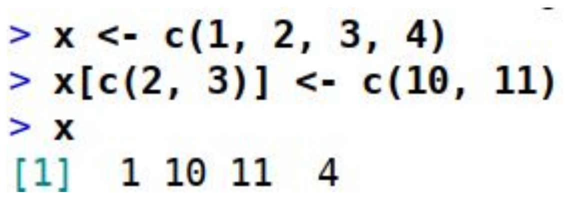
- `[ ]` makes it possible to assign multiple places in a vector at once:



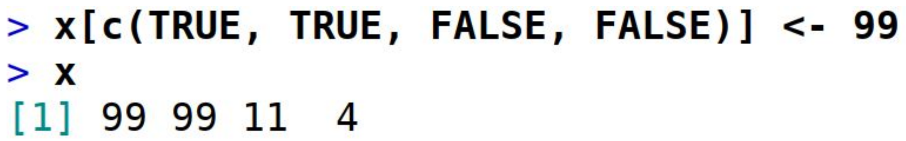
```
> x <- c(1, 2, 3, 4)
> x[c(2, 3)] <- c(10, 11)
> x
[1]  1 10 11  4
```

Reminder: Subset Assignment

- `[ ]` makes it possible to assign multiple places in a vector at once:



```
> x <- c(1, 2, 3, 4)
> x[c(2, 3)] <- c(10, 11)
> x
[1]  1 10 11  4
```



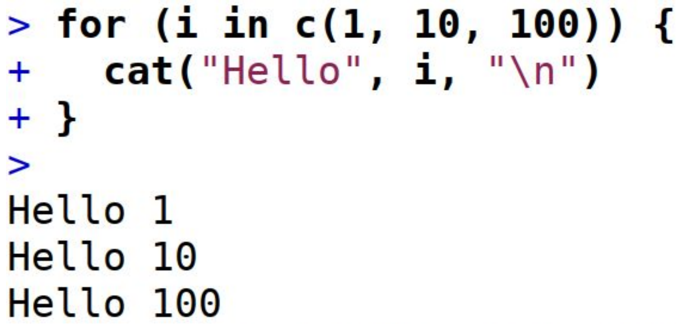
```
> x[c(TRUE, TRUE, FALSE, FALSE)] <- 99
> x
[1] 99 99 11  4
```

Reminder: Loops

- "for"-loop: `for (index in <vector/list>) { ... }`

Reminder: Loops

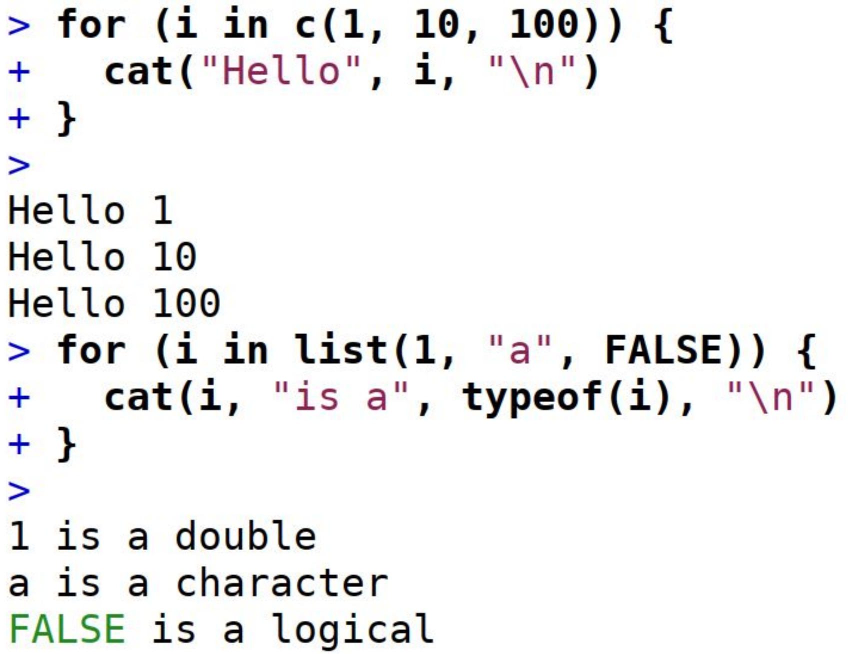
- "for"-loop: `for (index in <vector/list>) { ... }`
  ➔ Run content of `{ ... }` for every element in `<vector/list>`.



```
> for (i in c(1, 10, 100)) {
+   cat("Hello", i, "\n")
+ }
>
Hello 1
Hello 10
Hello 100
```

Reminder: Loops

- "for"-loop: `for (index in <vector/list>) { ... }`
  - ➔ Run content of `{ ... }` for every element in `<vector/list>`.



```
> for (i in c(1, 10, 100)) {
+   cat("Hello", i, "\n")
+ }
>
Hello 1
Hello 10
Hello 100
> for (i in list(1, "a", FALSE)) {
+   cat(i, "is a", typeof(i), "\n")
+ }
>
1 is a double
a is a character
FALSE is a logical
```

Lists also work!

Reminder: Loops

- "for"-loop: `for (index in <vector/list>) { ... }`

```
DO NOT use:
for (index in 1:number) { ... }
```

this has bad behaviour when `number` is `0`!

DO use:

```
for (index in seq_len(number)) { ... }
```

Reminder: Loops

- "for"-loop: `for (index in <vector/list>) { ... }`

```
DO NOT use:
for (index in 1:len(vector)) { ... }
```

this has bad behaviour when `vector` is `empty`!

DO use:

```
for (index in seq_along(vector)) { ... }
```

Reminder: Loops

- "for"-loop: `for (index in <vector/list>) { ... }`
- "while"-loop: `while (condition) { ... }`
- "repeat"-loop: infinite loop. `repeat { ... }`
  - The same as `while (TRUE) { ... }`

Control flow within loops:

- break out of loop: `break`
- continue to next round of the loop: `next`

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
    - You want to do something a given number of times

```
DO NOT use:
for (index in 1:number) { ... }
```

this has bad behaviour when `number` is `0`!

DO use:

```
for (index in seq_len(number)) { ... }
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
  - You want to do something a given number of times

```
DO NOT use:
for (index in 1:length(vector)) { ... }
```

this has bad behaviour when `vector` is `empty`!

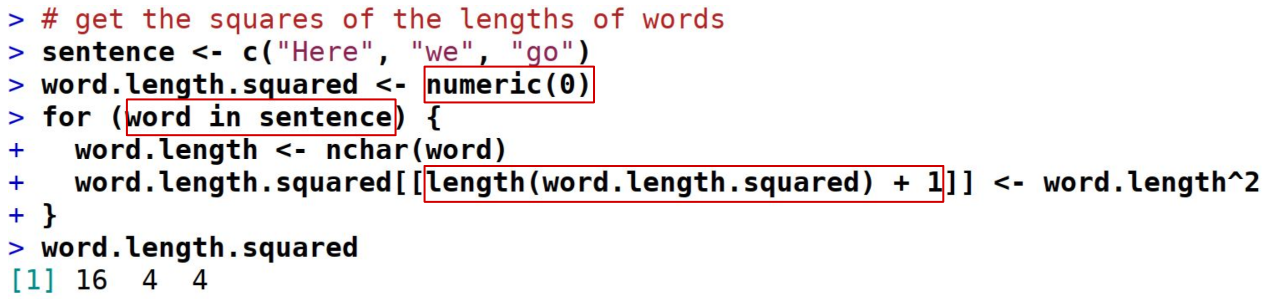
DO use:

```
for (index in seq_along(vector)) { ... }
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
  - You want to do something a given number of times
  - You have a vector and want to do an operation for every element, e.g.:
    - print it (see examples above)
    - do some computation and construct a new vector from it

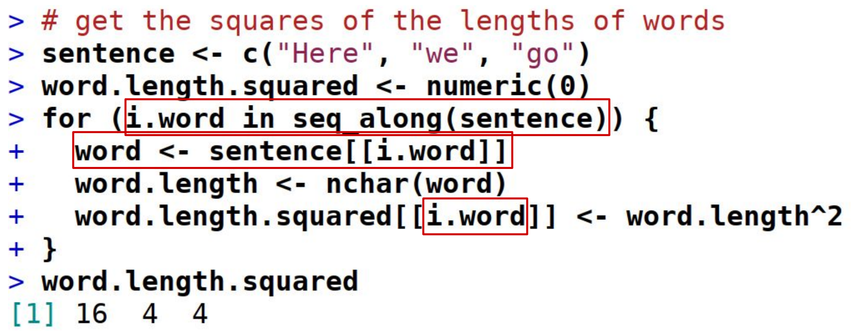


```
> # get the squares of the lengths of words
> sentence <- c("Here", "we", "go")
> word.length.squared <- numeric(0)
> for (word in sentence) {
+   word.length <- nchar(word)
+   word.length.squared[[length(word.length.squared) + 1]] <- word.length^2
+ }
> word.length.squared
[1] 16  4  4
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
  - You want to do something a given number of times
  - You have a vector and want to do an operation for every element, e.g.:
    - print it (see examples above)
    - do some computation and construct a new vector from it

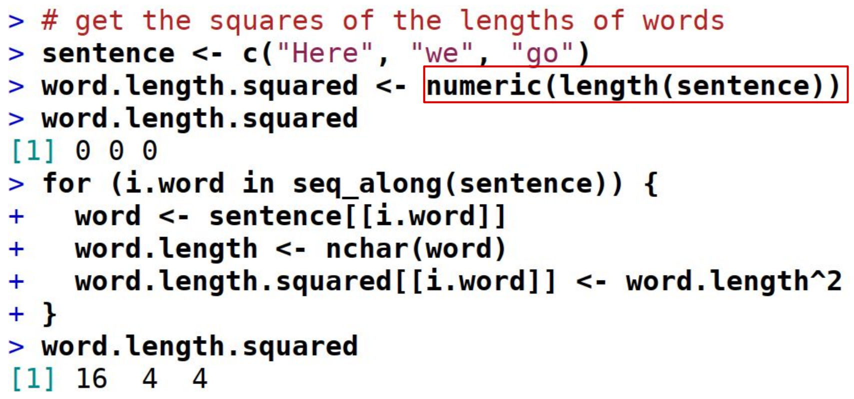


```
> # get the squares of the lengths of words
> sentence <- c("Here", "we", "go")
> word.length.squared <- numeric(0)
> for (i.word in seq_along(sentence)) {
+   word <- sentence[[i.word]]
+   word.length <- nchar(word)
+   word.length.squared[[i.word]] <- word.length^2
+ }
> word.length.squared
[1] 16  4  4
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
    - You want to do something a given number of times
    - You have a vector and want to do an operation for every element, e.g.:
        - print it (see examples above)
        - do some computation and construct a new vector from it



```
> # get the squares of the lengths of words
> sentence <- c("Here", "we", "go")
> word.length.squared <- numeric(length(sentence))
> word.length.squared
[1] 0 0 0
> for (i.word in seq_along(sentence)) {
+   word <- sentence[[i.word]]
+   word.length <- nchar(word)
+   word.length.squared[[i.word]] <- word.length^2
+ }
> word.length.squared
[1] 16  4  4
```

Reminder: Loops

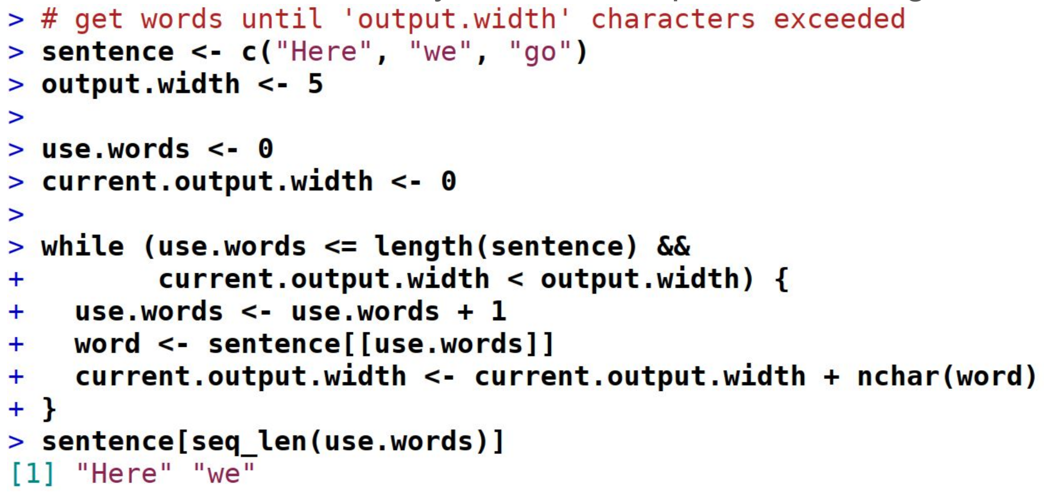
When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
    - You want to do something a given number of times
    - You have a vector and want to do an operation for every element, e.g.:
        - print it (see examples above)
        - do some computation and construct a new vector from it
    - You have a matrix / data.frame and want to do something for every row / column

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
- "while"-loop: `while (condition) { ... }`
  - You don't know how often you need to repeat something

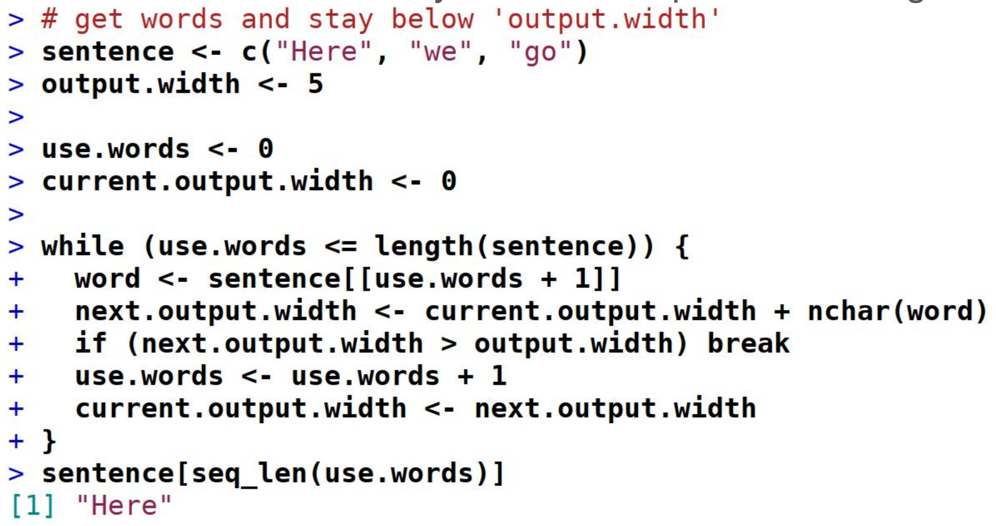


```
> # get words until 'output.width' characters exceeded
> sentence <- c("Here", "we", "go")
> output.width <- 5
>
> use.words <- 0
> current.output.width <- 0
>
> while (use.words <= length(sentence) &&
+        current.output.width < output.width) {
+   use.words <- use.words + 1
+   word <- sentence[[use.words]]
+   current.output.width <- current.output.width + nchar(word)
+ }
> sentence[seq_len(use.words)]
[1] "Here" "we"
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
- "while"-loop: `while (condition) { ... }`
    - You don't know how often you need to repeat something

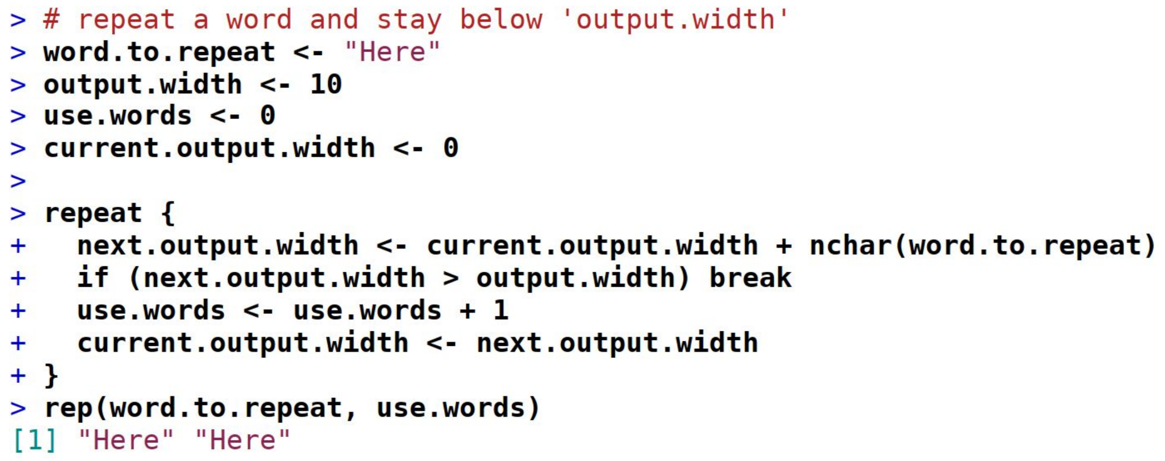


```
> # get words and stay below 'output.width'
> sentence <- c("Here", "we", "go")
> output.width <- 5
>
> use.words <- 0
> current.output.width <- 0
>
> while (use.words <= length(sentence)) {
+   word <- sentence[[use.words + 1]]
+   next.output.width <- current.output.width + nchar(word)
+   if (next.output.width > output.width) break
+   use.words <- use.words + 1
+   current.output.width <- next.output.width
+ }
> sentence[seq_len(use.words)]
[1] "Here"
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
- "while"-loop: `while (condition) { ... }`
- "repeat"-loop: infinite loop. `repeat { ... }`
  - Similar to while, but need to check in the middle of the loop whether to continue



```
> # repeat a word and stay below 'output.width'
> word.to.repeat <- "Here"
> output.width <- 10
> use.words <- 0
> current.output.width <- 0
>
> repeat {
+   next.output.width <- current.output.width + nchar(word.to.repeat)
+   if (next.output.width > output.width) break
+   use.words <- use.words + 1
+   current.output.width <- next.output.width
+ }
> rep(word.to.repeat, use.words)
[1] "Here" "Here"
```

Reminder: Loops

When to use what

- "for"-loop: `for (index in <vector/list>) { ... }`
- "while"-loop: `while (condition) { ... }`
- "repeat"-loop: infinite loop. `repeat { ... }`
    - Similar to while, but need to check in the middle of the loop whether to continue
    - Interactively: run something until the user stops it

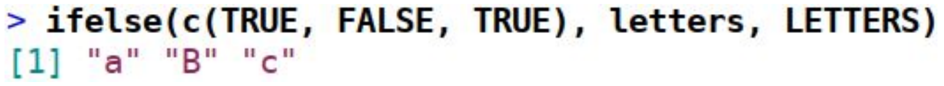
Reminder: Conditionals

- "if"-conditional: `if (condition) { ... } else { ... }`

Similar:

- ifelse-function: `ifelse(conditions, ..., ...)`

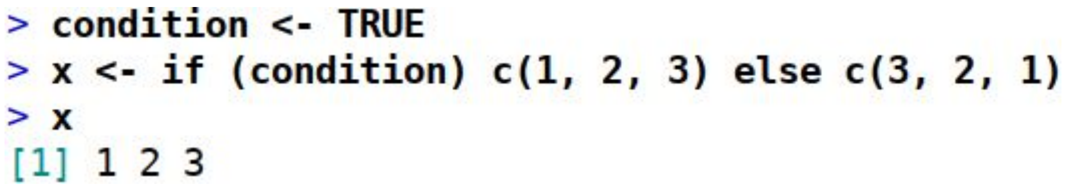
The difference: `ifelse` *vectorizes*:



```
> ifelse(c(TRUE, FALSE, TRUE), letters, LETTERS)
[1] "a" "B" "c"
```

=> use `ifelse` *only* if you have a *vector* of logicals.

=> use `if` otherwise. The following does *not* work with `ifelse`:



```
> condition <- TRUE
> x <- if (condition) c(1, 2, 3) else c(3, 2, 1)
> x
[1] 1 2 3
```

Reminder: Conditionals

Vectorization of Logicals

- Be aware: sometimes you want to work with *scalar truth values*, sometimes with *vector truth values*
  - Scalar truth values: e.g. `while`, `if`
  - Vector truth values: e.g. `ifelse`, vector indexing `x[c(TRUE, FALSE, TRUE)]`
- Different operators for them!
  - Operators for *vector truth values*: `|`, `&`.
  - Operators for *scalar truth values* `||`, `&&`.
- Be aware that `==` and similar gives *vector truth values* if comparing more than one value. use `all(x == y)`, `identical(x, y)` or `isTRUE(all.equal(x, y))` instead!

What We Expect You to Know

Structured Programming I

Know the material from the last lecture! In particular

- What the different types of vectors are, and what differentiates them from lists
- Different ways of accessing and setting elements in vectors, lists, matrices, and data.frames
- Vectorization and recycling by operators
- Loops: for, while, repeat, next, break
- Conditionals: if, else, switch(), ifelse()
- Logical operators, differences between &,| and &&,|| and when each is needed

What We Expect You to Know

Structured Programming I

Learn about these useful functions included in R that you may not have known before. `help()` is always a good start for this!

- Sequences (similar to colon operator): `seq_len, seq_along, seq`
- Sets: `setdiff, union, unique, setequal, duplicated, anyDuplicated, table`
- Indexing & finding things: `which, which.max, which.min, max, min, pmax, pmin, match, row, col`
- Logic operations: `all, any, identical, all.equal, isTRUE, isFALSE, xor`
- Value transformations: `cut, hist, diff, floor, ceiling, round, trunc`
- Vector reordering: `head, tail, append, rep, rep_len, rev, sort, order, sample`
- Type and type conversion: `is.<TYPE>, as.<TYPE>, anyNA, is.na, is.finite, mode, type, typeof, ordered, factor, unlist, class`
- Basic string operations: `nchar, substr, sprintf, toupper, tolower, paste, paste0`

Dev Track
Programming Style

Programming Style

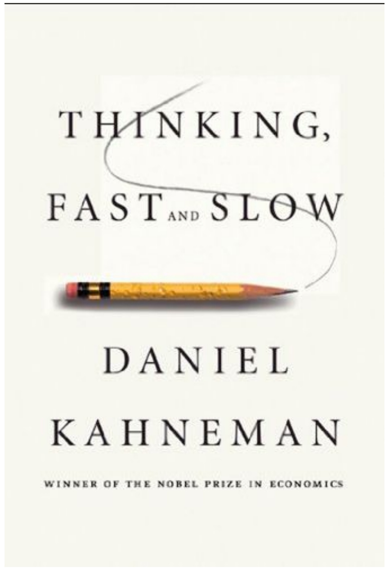
- As was already said in the first lecture, we not only want to teach you R itself, but also how to write software well.
- **Programming Style**: Suppose you know what sequence of commands you want R to execute.
  - How should you write these commands down?
  - What formatting do you use?
  - How do you break your code up into functions?
  - How do you name these functions?

Programming Style

- The program you write will not only be read by your computer, but also by humans:
  - Other people. Even if you are not (planning to) collaborate with others, eventually you will.
    - Unlike some physical tasks, programming naturally lends itself to division of labour. When you build a chair for yourself and your friend also wants a chair, you have to put in the same effort again. When you write a program to do your data analysis for your thesis, and your friend wants to do the same analysis, then he can just copy your code... and you better hope you wrote the program in an understandable way!
  - The future "you".
    - Look at your calendar from a year ago and try to remember what you meant with all the different one-word entries you wrote down in a hurry.
    - Future you in a long enough time is essentially a stranger who will hate you if they can't read your code.
- Adapt your code to be readable and easily adaptable!

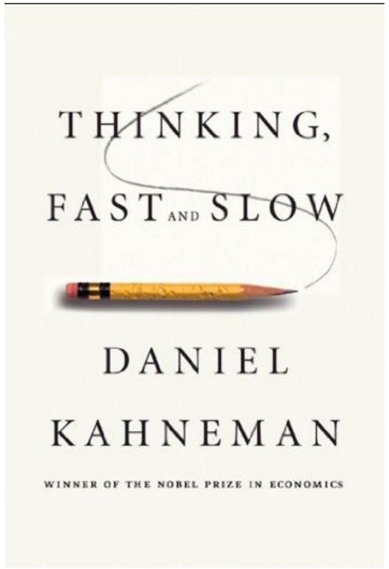
A Short Detour to Human Psychology

- Kahneman: Thinking, Fast and Slow
  - "System 1": emotional, stereotypic, unconscious, going by first appearances, fast, automatic
  - "System 2": logical, calculating, conscious effort, slow, effortful



A Short Detour to Human Psychology

- Kahneman: Thinking, Fast and Slow
  - "System 1": emotional, stereotypic, unconscious, going by first appearances, fast, automatic
  - "System 2": **logical, calculating**, conscious effort, slow, effortful
- Computers do "System 2"!
  - does not care about what your code "looks like"
  - there is no ambiguity, just does the calculation

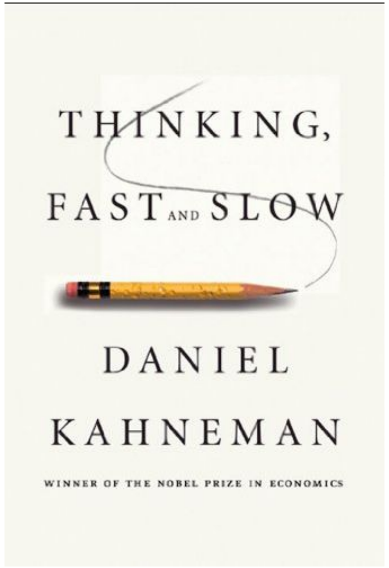


A Short Detour to Human Psychology

- Kahneman: Thinking, Fast and Slow
  - "System 1": emotional, stereotypic, unconscious, going by first appearances, fast, automatic
  - "System 2": **logical, calculating**, conscious effort, slow, effortful
- Computers do "System 2"!
  - does not care about what your code "looks like"
  - there is no ambiguity, just does the calculation

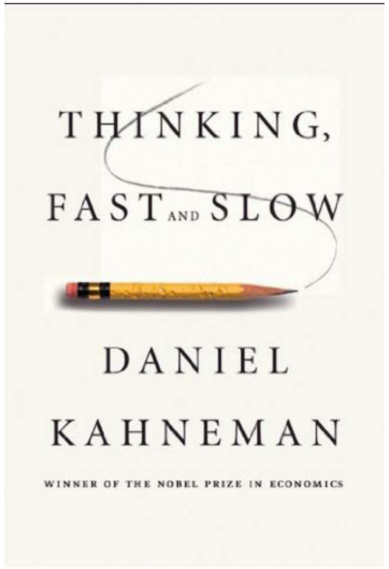
Computers have no problems with this:

```
f <- (function(f) (function(x) f(x(x)))(function(x) f(x(x))))(
  function(x) function(n) if (n < 1) 1 else n * x(n - 1))
```



A Short Detour to Human Psychology

- Kahneman: Thinking, Fast and Slow
  - "System 1": emotional, stereotypic, unconscious, going by first appearances, **fast, automatic**
  - "System 2": logical, calculating, conscious effort, slow, effortful
- Computers do "System 2"!
  - does not care about what your code "looks like"
  - there is no ambiguity, just does the calculation
- Humans prefer "System 1"!
  - judge by appearances and familiarity

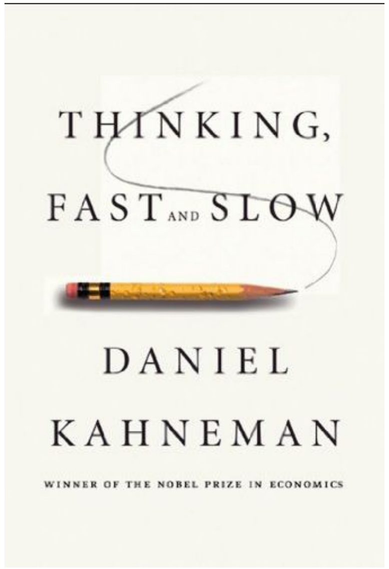


A Short Detour to Human Psychology

- Kahneman: Thinking, Fast and Slow
  - "System 1": emotional, stereotypic, unconscious, going by first appearances, **fast, automatic**
  - "System 2": logical, calculating, conscious effort, slow, effortful
- Computers do "System 2"!
  - does not care about what your code "looks like"
  - there is no ambiguity, just does the calculation
- Humans prefer "System 1"!
  - judge by appearances and familiarity

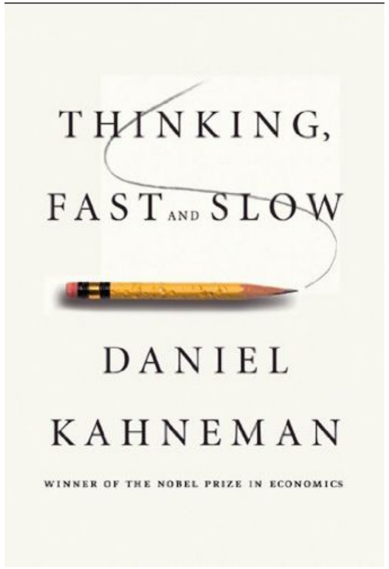
equivalent to the function two slides before:

```
factorial <- function(n) {
  if (n < 1) {
    return(1)
  } else {
    return(n * factorial(n - 1))
  }
}
```



A Short Detour to Human Psychology

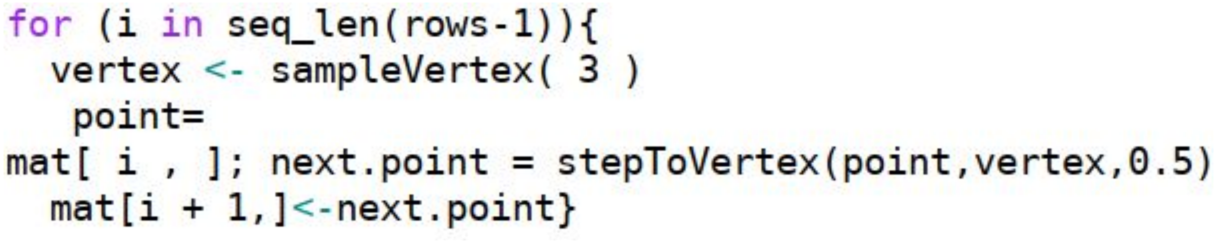
- Kahneman: Thinking, Fast and Slow
  - "System 1": emotional, stereotypic, unconscious, going by first appearances, fast, automatic
  - "System 2": logical, calculating, conscious effort, **slow, effortful**
- Computers do "System 2"!
  - does not care about what your code "looks like"
  - there is no ambiguity, just does the calculation
- Humans prefer "System 1"!
  - judge by appearances and familiarity
- Your program must satisfy both: System 2 (or the computer will not do what you want) and System 1 (Or it will be slow and effortful to understand what is happening)



Programming Style (I):
Avoid Unnecessary Noise & Variation

Programming Style

Avoid unnecessary Noise & Variation

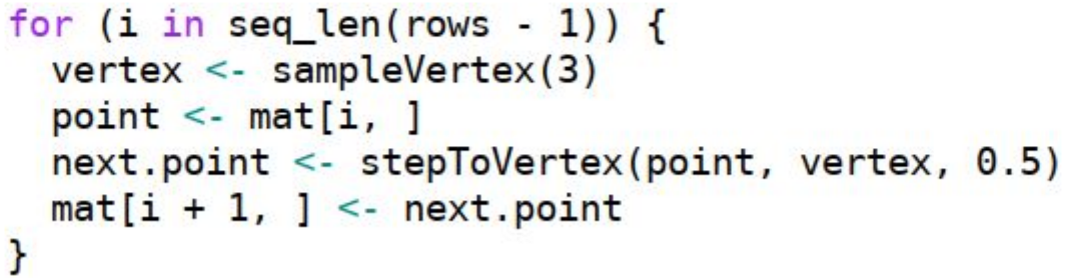


```
for (i in seq_len(rows-1)){
  vertex <- sampleVertex( 3 )
   point=
mat[ i , ]; next.point = stepToVertex(point,vertex,0.5)
  mat[i + 1,]<-next.point}
```

- Come on, it's just ugly. Have some pride in your work!

Programming Style

Avoid unnecessary Noise & Variation

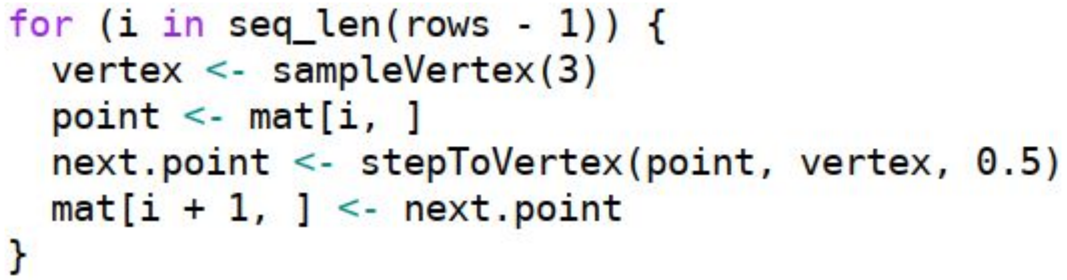


```r
for (i in seq_len(rows - 1)) {
  vertex <- sampleVertex(3)
  point <- mat[i, ]
  next.point <- stepToVertex(point, vertex, 0.5)
  mat[i + 1, ] <- next.point
}
```

- Much more regular!

Programming Style

Avoid unnecessary Noise & Variation



```
for (i in seq_len(rows - 1)) {
  vertex <- sampleVertex(3)
  point <- mat[i, ]
  next.point <- stepToVertex(point, vertex, 0.5)
  mat[i + 1, ] <- next.point
}
```

- don't write `a+b` at one point and `a + b` at another etc.
- this is sometimes referred to as "**code style**" in the narrower sense

Programming Style: Code Style

- where to put your parentheses (), braces {}, brackets []
- where to put your spaces, and how many of them
- how to name your variables and functions

Programming Style: Code Style

- where to put your parentheses (), braces {}, brackets []
- where to put your spaces, and how many of them
- how to name your variables and functions

e.g.

```
if (x == 1) {
  y <- 2
}
```

vs.

```
if(x==1)
      {
            y=2
      }
```

Programming Style: Code Style

- where to put your parentheses (), braces {}, brackets []
- where to put your spaces, and how many of them
- how to name your variables and functions

e.g.

```
if (x == 1) {
  y <- 2
}
```

vs.

```
if(x==1)
     {
          y=2
     }
```

- could both be fine as long as you are consistent
- should be chosen for easy readability, but to some degree a matter of aesthetic preferences

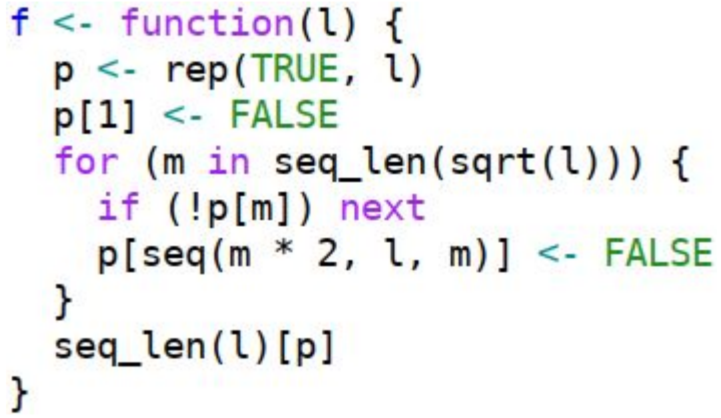
Programming Style (II): Use All Communication Channels Available to You

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names

  Not so nice: single letters

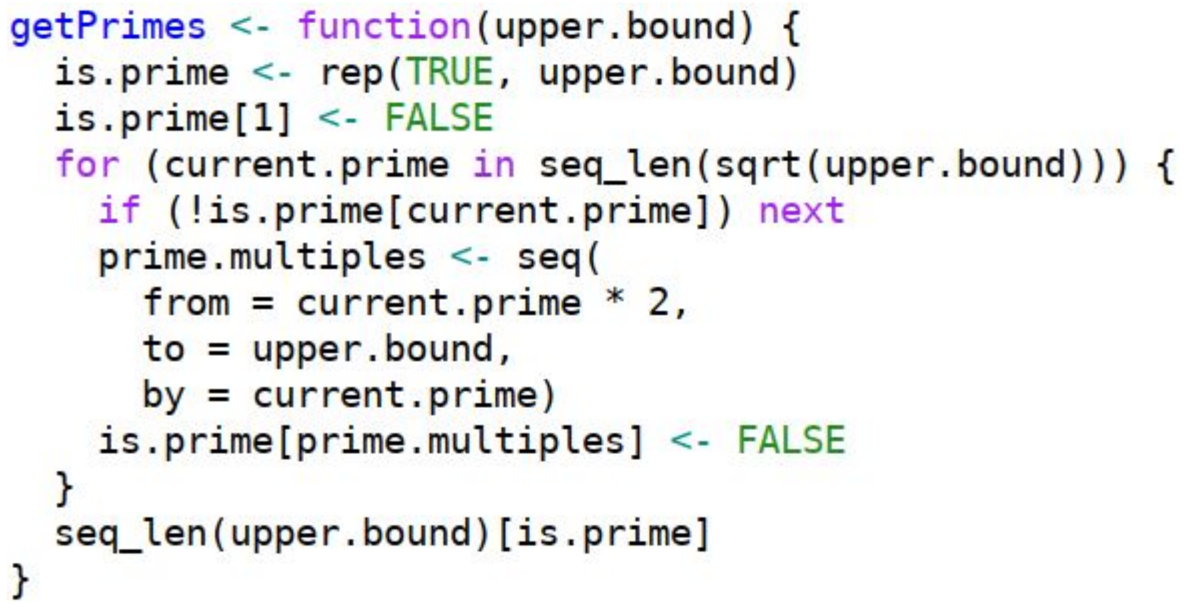


```r
f <- function(l) {
  p <- rep(TRUE, l)
  p[1] <- FALSE
  for (m in seq_len(sqrt(l))) {
    if (!p[m]) next
    p[seq(m * 2, l, m)] <- FALSE
  }
  seq_len(l)[p]
}
```

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names



```r
getPrimes <- function(upper.bound) {
  is.prime <- rep(TRUE, upper.bound)
  is.prime[1] <- FALSE
  for (current.prime in seq_len(sqrt(upper.bound))) {
    if (!is.prime[current.prime]) next
    prime.multiples <- seq(
      from = current.prime * 2,
      to = upper.bound,
      by = current.prime)
    is.prime[prime.multiples] <- FALSE
  }
  seq_len(upper.bound)[is.prime]
}
```

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names

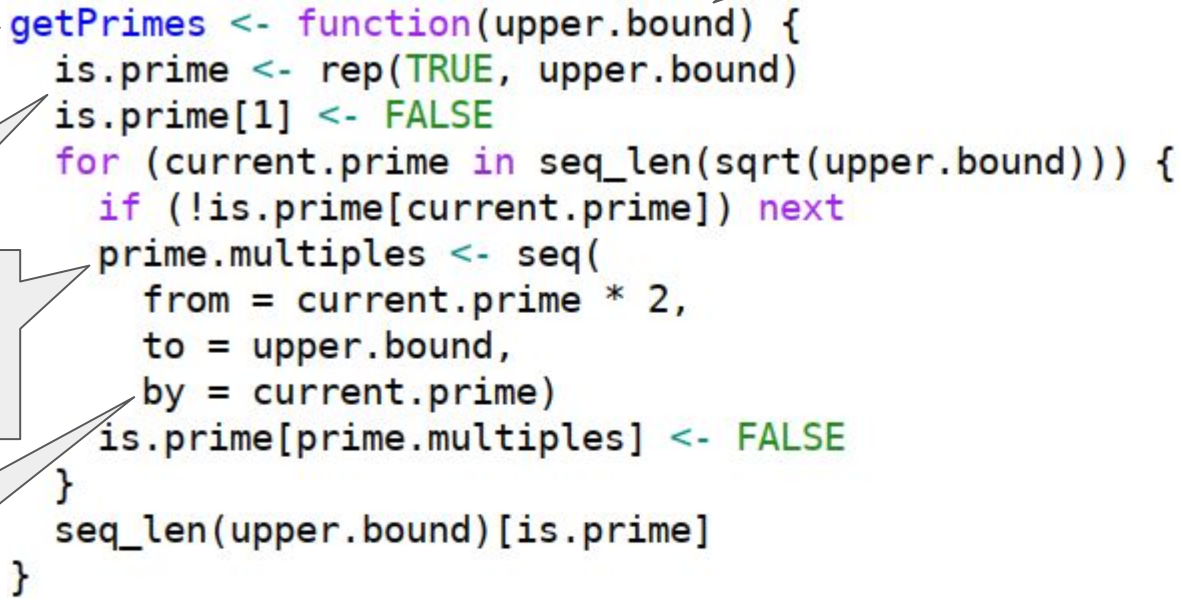
Descriptive argument name

Descriptive function name

English. Always. No exceptions.

Introduce temporary variable we didn't have before for clarity

call less-well-known positional function arguments by name



```r
getPrimes <- function(upper.bound) {
  is.prime <- rep(TRUE, upper.bound)
  is.prime[1] <- FALSE
  for (current.prime in seq_len(sqrt(upper.bound))) {
    if (!is.prime[current.prime]) next
    prime.multiples <- seq(
      from = current.prime * 2,
      to = upper.bound,
      by = current.prime)
    is.prime[prime.multiples] <- FALSE
  }
  seq_len(upper.bound)[is.prime]
}
```

Programming Style

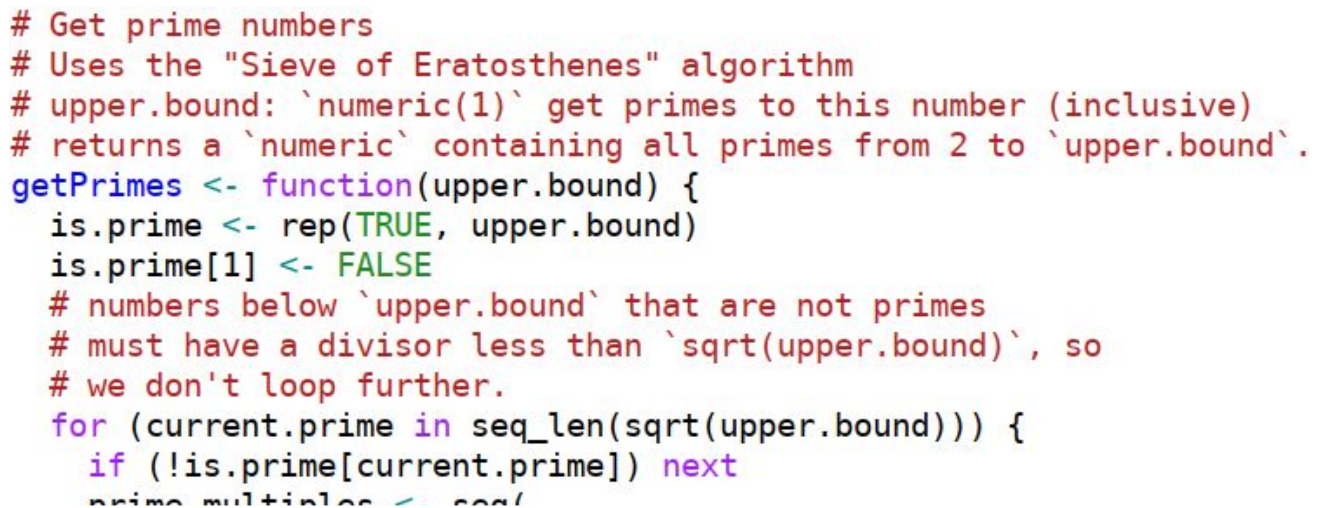
Use All Communication Channels Available to You

- Use descriptive variable / function names
  - Some further notes:
    - short variable names are fine if they are established practice. E.g. "`i`" as iterator in for-loops, "`tmp`" for temporary values.
    - avoid collisions with functions in base R or common packages. Don't call your function "`mean`" or "`ggplot`". Don't call your variable "`c`" because of collision with `c(1, 2)`!
    - avoid names that look similar, e.g. only differentiated through `1` (one) vs. `l` (lowercase L) vs. `I` (uppercase i). Even if *your* system has a font that distinguishes those, someone else's might not.
    - Keep with a pattern. The pendant to "`coord.x`" should *not* be "`y.coordinate`"
    - Consider tab-completion when naming things: names should be `<general>.<special>`. "`coord.x`" is preferred, because if someone else is editing the program and wants to get a coordinate, they enter `coord.`<tab> and can see what coordinates are available.

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments!



```r
# Get prime numbers
# Uses the "Sieve of Eratosthenes" algorithm
# upper.bound: `numeric(1)` get primes to this number (inclusive)
# returns a `numeric` containing all primes from 2 to `upper.bound`.
getPrimes <- function(upper.bound) {
  is.prime <- rep(TRUE, upper.bound)
  is.prime[1] <- FALSE
  # numbers below `upper.bound` that are not primes
  # must have a divisor less than `sqrt(upper.bound)`, so
  # we don't loop further.
  for (current.prime in seq_len(sqrt(upper.bound))) {
    if (!is.prime[current.prime]) next
    prime.multiples <- seq(
```

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments!

What does the function do?

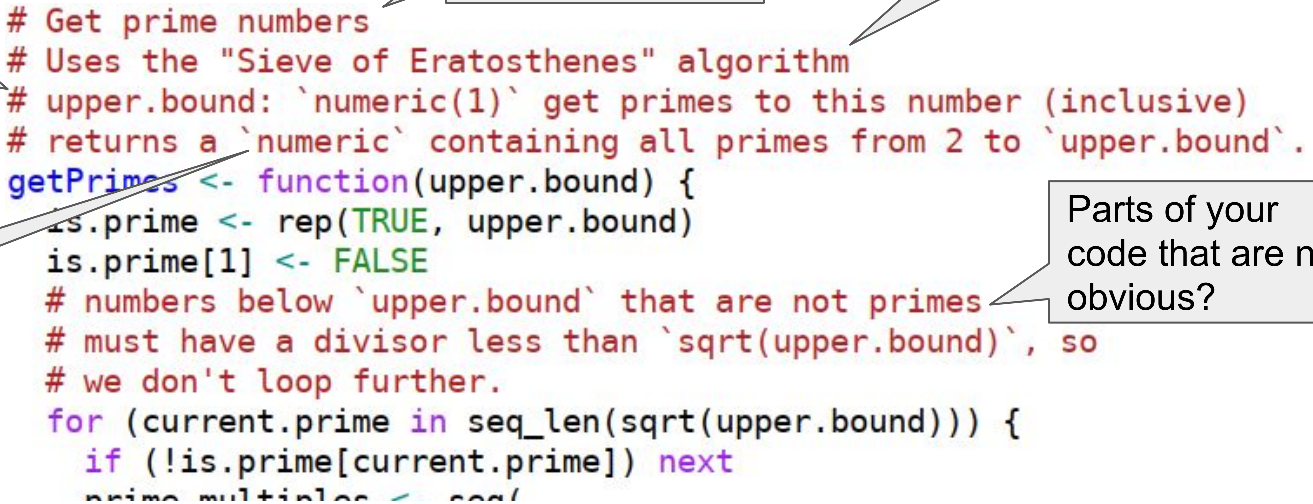
English. Always. No exceptions.

What are its arguments?

What is the return value?

What are the argument / return **types**?

Parts of your code that are not obvious?



```r
# Get prime numbers
# Uses the "Sieve of Eratosthenes" algorithm
# upper.bound: `numeric(1)` get primes to this number (inclusive)
# returns a `numeric` containing all primes from 2 to `upper.bound`.
getPrimes <- function(upper.bound) {
  is.prime <- rep(TRUE, upper.bound)
  is.prime[1] <- FALSE
  # numbers below `upper.bound` that are not primes
  # must have a divisor less than `sqrt(upper.bound)`, so
  # we don't loop further.
  for (current.prime in seq_len(sqrt(upper.bound))) {
    if (!is.prime[current.prime]) next
    prime.multiples <- seq(
```

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments! (**but don't overdo it**)

```r
~~# we get a vector of `TRUE` with length `upper.bound`~~
is.prime <- rep(TRUE, upper.bound)
~~# `1` is not a prime~~
is.prime[1] <- FALSE
```

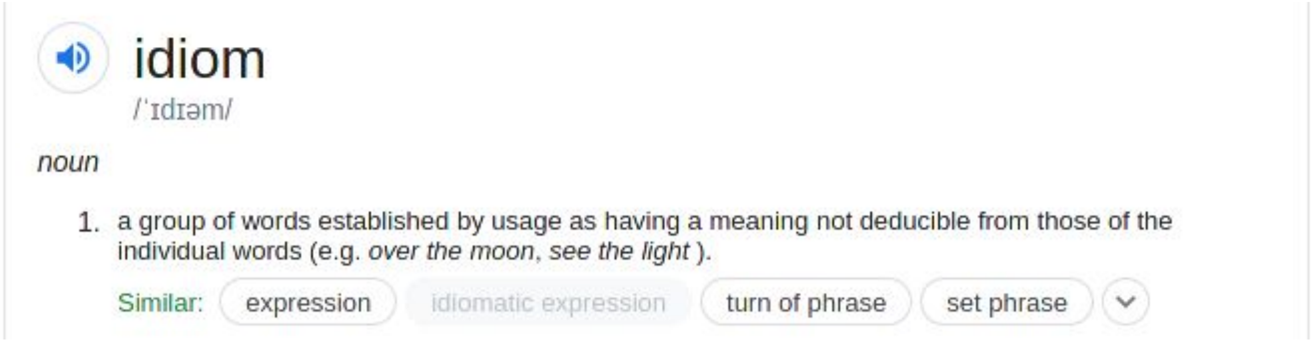
don't just translate the code to english

what happens here is obvious from good naming already

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments! (but don't overdo it)
- Make your code **idiomatic**



idiom

/ˈɪdɪəm/

noun

1. a group of words established by usage as having a meaning not deducible from those of the individual words (e.g. *over the moon, see the light*).

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments! (but don't overdo it)
- Make your code **idiomatic**
    - express it in a natural and commonly accepted way specific to the language

~~`seq_len(upper.bound)[is.prime]`~~

"what kind of trick is this?"

"I've seen this before!"

`which(is.prime)`

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments! (but don't overdo it)
- Make your code **idiomatic**
  - express it in a natural and commonly accepted way specific to the language
  - Don't be "too clever"
  - Use constructs that the reader of your code is used to from other code
  - Solve problems the way someone would expect them to be solved in R

Programming Style

Use All Communication Channels Available to You

- Use descriptive variable / function names
- Use comments! (but don't overdo it)
- Make your code **idiomatic**
  - express it in a natural and commonly accepted way specific to the language
  - Don't be "too clever"
  - Use constructs that the reader of your code is used to from other code
  - Solve problems the way someone would expect them to be solved in R
  - **This takes experience**, you will learn common idioms by reading other code

Programming Style (III):
Your Audience has Tunnel Vision

Programming Style

Your Audience has Tunnel Vision

When you spent all day working on a function of course you know what is happening, but someone else should not have to spend a day to understand it

Programming Style

Your Audience has Tunnel Vision

Your code should be understandable

- without needing to see many lines at once

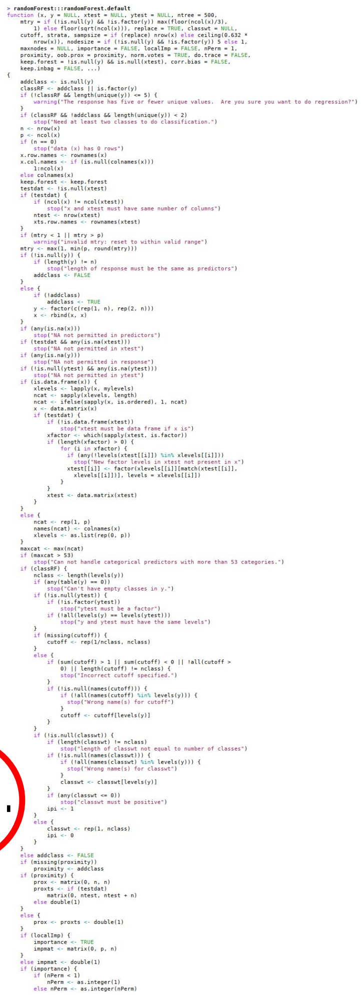
Programming Style

Your Audience has Tunnel Vision

Your code should be understandable

- without needing to see many lines at once

this is *half* of the `randomForest()` function



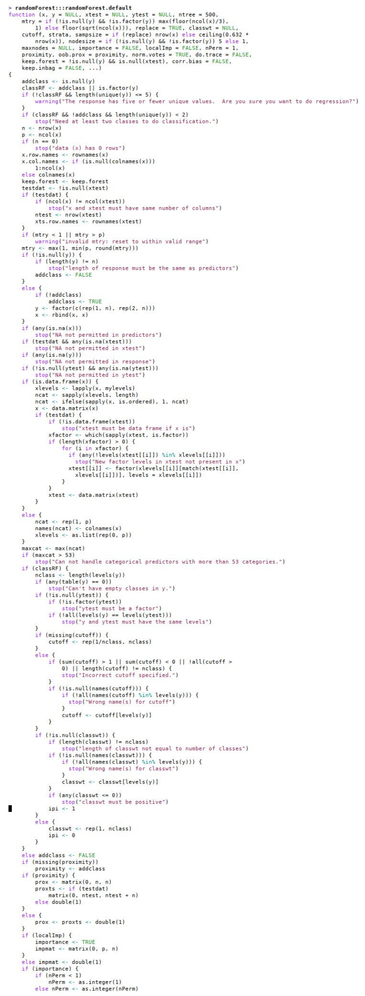
```r
> randomForest:::randomForest.default
function (x, y = NULL, xtest = NULL, ytest = NULL, ntree = 500,
    mtry = if (!is.null(y) && !is.factor(y)) max(floor(ncol(x)/3),
        1) else floor(sqrt(ncol(x))), replace = TRUE, classwt = NULL,
    cutoff, strata, sampsize = if (replace) nrow(x) else ceiling(0.632 *
        nrow(x)), nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
    maxnodes = NULL, importance = FALSE, localImp = FALSE, nPerm = 1,
    proximity, oob.prox = proximity, norm.votes = TRUE, do.trace = FALSE,
    keep.forest = !is.null(y) && is.null(xtest), corr.bias = FALSE,
    keep.inbag = FALSE, ...)
{
    addclass <- is.null(y)
    classRF <- addclass || is.factor(y)
    if (!classRF && length(unique(y)) <= 5) {
        warning("The response has five or fewer unique values.  Are you sure you want to do regression?")
    }
    if (classRF && !addclass && length(unique(y)) < 2)
        stop("Need at least two classes to do classification.")
    n <- nrow(x)
    p <- ncol(x)
    if (n == 0)
        stop("data (x) has 0 rows")
    x.row.names <- rownames(x)
    x.col.names <- if (is.null(colnames(x)))
        1:ncol(x)
    else colnames(x)
    keep.forest <- keep.forest
    testdat <- !is.null(xtest)
    if (testdat) {
        if (ncol(x) != ncol(xtest))
            stop("x and xtest must have same number of columns")
        ntest <- nrow(xtest)
        xts.row.names <- rownames(xtest)
    }
    if (mtry < 1 || mtry > p)
        warning("invalid mtry: reset to within valid range")
    mtry <- max(1, min(p, round(mtry)))
    if (!is.null(y)) {
        if (length(y) != n)
            stop("length of response must be the same as predictors")
        addclass <- FALSE
    }
    else {
        if (!addclass)
            addclass <- TRUE
        y <- factor(c(rep(1, n), rep(2, n)))
        x <- rbind(x, x)
    }
    if (any(is.na(x)))
        stop("NA not permitted in predictors")
    if (testdat && any(is.na(xtest)))
        stop("NA not permitted in xtest")
    if (any(is.na(y)))
        stop("NA not permitted in response")
    if (!is.null(ytest) && any(is.na(ytest)))
        stop("NA not permitted in ytest")
    if (is.data.frame(x)) {
        xlevels <- lapply(x, mylevels)
        ncat <- sapply(xlevels, length)
        ncat <- ifelse(sapply(x, is.ordered), 1, ncat)
        x <- data.matrix(x)
        if (testdat) {
            if (!is.data.frame(xtest))
                stop("xtest must be data frame if x is")
            xfactor <- which(sapply(xtest, is.factor))
            if (length(xfactor) > 0) {
                for (i in xfactor) {
                  if (any(!levels(xtest[[i]]) %in% xlevels[[i]]))
                    stop("New factor levels in xtest not present in x")
                  xtest[[i]] <- factor(xlevels[[i]][match(xtest[[i]],
                    xlevels[[i]])], levels = xlevels[[i]])
                }
            }
            xtest <- data.matrix(xtest)
        }
    }
    else {
        ncat <- rep(1, p)
        names(ncat) <- colnames(x)
        xlevels <- as.list(rep(0, p))
    }
    maxcat <- max(ncat)
    if (maxcat > 53)
        stop("Can not handle categorical predictors with more than 53 categories.")
    if (classRF) {
        nclass <- length(levels(y))
        if (any(table(y) == 0))
            stop("Can't have empty classes in y.")
        if (!is.null(ytest)) {
            if (!is.factor(ytest))
                stop("ytest must be a factor")
            if (!all(levels(y) == levels(ytest)))
                stop("y and ytest must have the same levels")
        }
        if (missing(cutoff)) {
            cutoff <- rep(1/nclass, nclass)
        }
        else {
            if (sum(cutoff) > 1 || sum(cutoff) < 0 || !all(cutoff >
                0) || length(cutoff) != nclass) {
                stop("Incorrect cutoff specified.")
            }
            if (!is.null(names(cutoff))) {
                if (!all(names(cutoff) %in% levels(y))) {
                  stop("Wrong name(s) for cutoff")
                }
                cutoff <- cutoff[levels(y)]
            }
        }
        if (!is.null(classwt)) {
            if (length(classwt) != nclass)
                stop("length of classwt not equal to number of classes")
            if (!is.null(names(classwt))) {
                if (!all(names(classwt) %in% levels(y))) {
                  stop("Wrong name(s) for classwt")
                }
                classwt <- classwt[levels(y)]
            }
            if (any(classwt <= 0))
                stop("classwt must be positive")
            ipi <- 1
        }
        else {
            classwt <- rep(1, nclass)
            ipi <- 0
        }
    }
    else addclass <- FALSE
    if (missing(proximity))
        proximity <- addclass
    if (proximity) {
        prox <- matrix(0, n, n)
        proxts <- if (testdat)
            matrix(0, ntest, ntest + n)
        else double(1)
    }
    else {
        prox <- proxts <- double(1)
    }
    if (localImp) {
        importance <- TRUE
        impmat <- matrix(0, p, n)
    }
    else impmat <- double(1)
    if (importance) {
        if (nPerm < 1)
            nPerm <- as.integer(1)
        else nPerm <- as.integer(nPerm)
```

Programming Style

Your Audience has Tunnel Vision

Your code should be understandable

- without needing to see many lines at once
  - chunk your code into reasonably sized functions
    - should ideally fit on a screen
  - and reasonably sized files



```
> randomForest:::randomForest.default
function (x, y = NULL, xtest = NULL, ytest = NULL, ntree = 500,
    mtry = if (!is.null(y) && !is.factor(y)) max(floor(ncol(x)/3),
        1) else floor(sqrt(ncol(x))), replace = TRUE, classwt = NULL,
    cutoff, strata, sampsize = if (replace) nrow(x) else ceiling(0.632 *
        nrow(x)), nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
    maxnodes = NULL, importance = FALSE, localImp = FALSE, nPerm = 1,
    proximity, oob.prox = proximity, norm.votes = TRUE, do.trace = FALSE,
    keep.forest = !is.null(y) && is.null(xtest), corr.bias = FALSE,
    keep.inbag = FALSE, ...)
{
    addclass <- is.null(y)
    classRF <- addclass || is.factor(y)
    if (!classRF && length(unique(y)) <= 5) {
        warning("The response has five or fewer unique values.  Are you sure you want to do regression?")
    }
    if (classRF && !addclass && length(unique(y)) < 2)
        stop("Need at least two classes to do classification.")
    n <- nrow(x)
    p <- ncol(x)
    if (n == 0)
        stop("data (x) has 0 rows")
    x.row.names <- rownames(x)
    x.col.names <- if (is.null(colnames(x)))
        1:ncol(x)
    else colnames(x)
    keep.forest <- keep.forest
    testdat <- !is.null(xtest)
    if (testdat) {
        if (ncol(x) != ncol(xtest))
            stop("x and xtest must have same number of columns")
        ntest <- nrow(xtest)
        xts.row.names <- rownames(xtest)
    }
    if (mtry < 1 || mtry > p)
        warning("invalid mtry: reset to within valid range")
    mtry <- max(1, min(p, round(mtry)))
    if (!is.null(y)) {
        if (length(y) != n)
            stop("length of response must be the same as predictors")
        addclass <- FALSE
    }
    else {
        if (!addclass)
            addclass <- TRUE
        y <- factor(c(rep(1, n), rep(2, n)))
        x <- rbind(x, x)
    }
    if (any(is.na(x)))
        stop("NA not permitted in predictors")
    if (testdat && any(is.na(xtest)))
        stop("NA not permitted in xtest")
    if (any(is.na(y)))
        stop("NA not permitted in response")
    if (!is.null(ytest) && any(is.na(ytest)))
        stop("NA not permitted in ytest")
    if (is.data.frame(x)) {
        xlevels <- lapply(x, mylevels)
        ncat <- sapply(xlevels, length)
        ncat <- ifelse(sapply(x, is.ordered), 1, ncat)
        x <- data.matrix(x)
        if (testdat) {
            if (!is.data.frame(xtest))
                stop("xtest must be data frame if x is")
            xfactor <- which(sapply(xtest, is.factor))
            if (length(xfactor) > 0) {
                for (i in xfactor) {
                  if (any(!levels(xtest[[i]]) %in% xlevels[[i]]))
                    stop("New factor levels in xtest not present in x")
                  xtest[[i]] <- factor(xlevels[[i]][match(xtest[[i]],
                    xlevels[[i]])], levels = xlevels[[i]])
                }
            }
            xtest <- data.matrix(xtest)
        }
    }
    else {
        ncat <- rep(1, p)
        names(ncat) <- colnames(x)
        xlevels <- as.list(rep(0, p))
    }
    maxcat <- max(ncat)
    if (maxcat > 53)
        stop("Can not handle categorical predictors with more than 53 categories.")
    if (classRF) {
        nclass <- length(levels(y))
        if (any(table(y) == 0))
            stop("Can't have empty classes in y.")
        if (!is.null(ytest)) {
            if (!is.factor(ytest))
                stop("ytest must be a factor")
            if (!all(levels(y) == levels(ytest)))
                stop("y and ytest must have the same levels")
        }
        if (missing(cutoff)) {
            cutoff <- rep(1/nclass, nclass)
        }
        else {
            if (sum(cutoff) > 1 || sum(cutoff) < 0 || !all(cutoff >
                0) || length(cutoff) != nclass) {
                stop("Incorrect cutoff specified.")
            }
            if (!is.null(names(cutoff))) {
                if (!all(names(cutoff) %in% levels(y))) {
                  stop("Wrong name(s) for cutoff")
                }
                cutoff <- cutoff[levels(y)]
            }
        }
        if (!is.null(classwt)) {
            if (length(classwt) != nclass)
                stop("length of classwt not equal to number of classes")
            if (!is.null(names(classwt))) {
                if (!all(names(classwt) %in% levels(y))) {
                  stop("Wrong name(s) for classwt")
                }
                classwt <- classwt[levels(y)]
            }
            if (any(classwt <= 0))
                stop("classwt must be positive")
            ipi <- 1
        }
        else {
            classwt <- rep(1, nclass)
            ipi <- 0
        }
    }
    else addclass <- FALSE
    if (missing(proximity))
        proximity <- addclass
    if (proximity) {
        prox <- matrix(0, n, n)
        proxts <- if (testdat)
            matrix(0, ntest, ntest + n)
        else double(1)
    }
    else {
        prox <- proxts <- double(1)
    }
    if (localImp) {
        importance <- TRUE
        impmat <- matrix(0, p, n)
    }
    else impmat <- double(1)
    if (importance) {
        if (nPerm < 1)
            nPerm <- as.integer(1)
        else nPerm <- as.integer(nPerm)
```

Programming Style

Your Audience has Tunnel Vision

Your code should be understandable

- without needing to see many lines at once
  - chunk your code into reasonably sized functions
    - should ideally fit on a screen
  - and reasonably sized files
- without having to keep many things in mind at once

Programming Style

Your Audience has Tunnel Vision

- without needing to see many lines at once
  - chunk your code into reasonably sized functions
    - should ideally fit on a screen
  - and reasonably sized files
- without having to keep many things in mind at once
  - limit your "nesting depth": avoid for-loops inside if-clauses inside for-loops inside ...
  - limit the number of args of functions. The user shouldn't have to check the docs all the time.
  - limit the role fulfilled by a single function. If you can't summarize its effect in one sentence, consider splitting it up.
  - related but more general: strive for **loose coupling**, i.e. limit the interdependencies between parts of your code and the degree to which one part depends on specific implementation details of another part.

Programming Style in ProgR

Programming Style in ProgR

- In this course, your code style will be **checked automatically**.
    - Your code deviates from what we dictate --> no points
    - But don't worry, the checker will give you informative feedback!

Programming Style in ProgR

- In this course, your code style will be **checked automatically**.
    - Your code deviates from what we dictate --> no points
    - But don't worry, the checker will give you informative feedback!
- We **mostly** use Google's (old) style guide:
    - Well written, short and to the point without being ambiguous
    - We add the **following alterations**:
        - Function naming: use `lowerCamel` for *functions* and `UpperCamel` for *classes* (you will see classes in a few weeks)
        - Line length: limit to 120 characters instead of 80. Welcome to the future!
        - Indentation: indent one level deeper per `{`, `(` or `[` one level up per `}`, `)` or `]`, except if they are opened *and* closed on the same line. Lines starting with closing parens get the reduced indentation level.
        - Don't use `return()` at the end of functions if you don't need to
        - Put comments describing the actions of a function *before* the function, to be consistent with roxygen2 which we will see later in the course.
        - Plus some good practices that you will find out about when checking your answers :-)

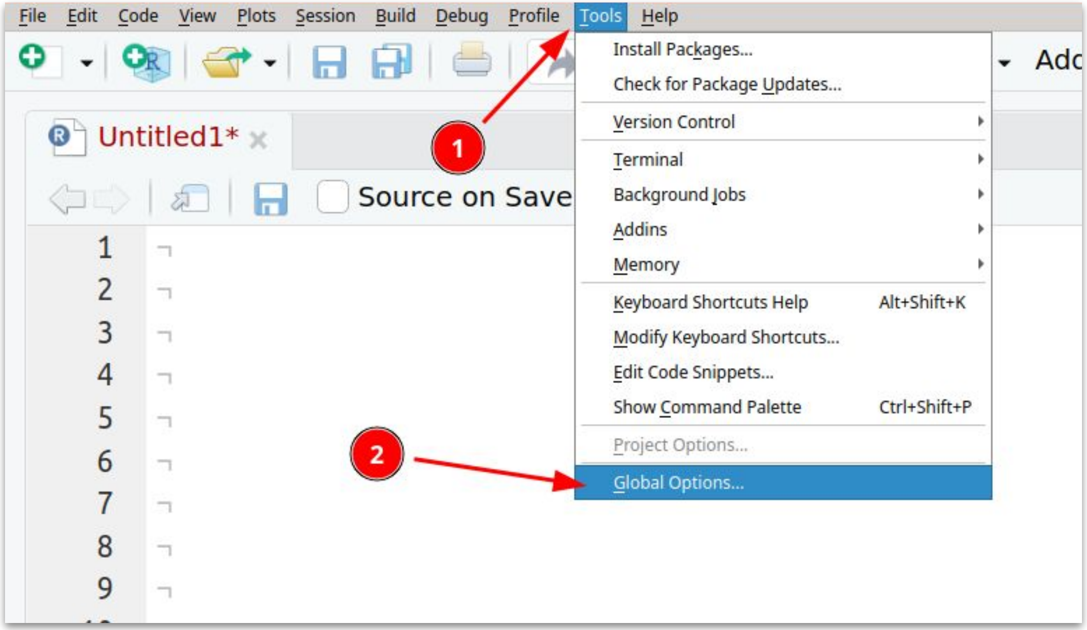
Programming Style in ProgR

- Notable other style guides:
  - "tidyverse" style guide.
    - Mostly similar, but much more to read. Largely concerned with "tidyverse"-internal extensions to R that are a bit niche.
    - Notable difference to what we do: they use `snake_case`, while we avoid underscores in variable and function names
  - Style of the "mlr"-package
    - Not that notable in the wide scheme of things, but the Bischl group at LMU uses it, so you will probably see it at some point.
    - Prominent differences to what we (and most other people) do: uses "`=`" instead of "`<-`" (which has some drawbacks). Make integers explicit with the "`L`" suffix.
- If this interests you, there is a study on prevalence of different styles in the R community by Chia-Yi Yen et al.

Programming Style in ProgR

Recommended RStudio Settings

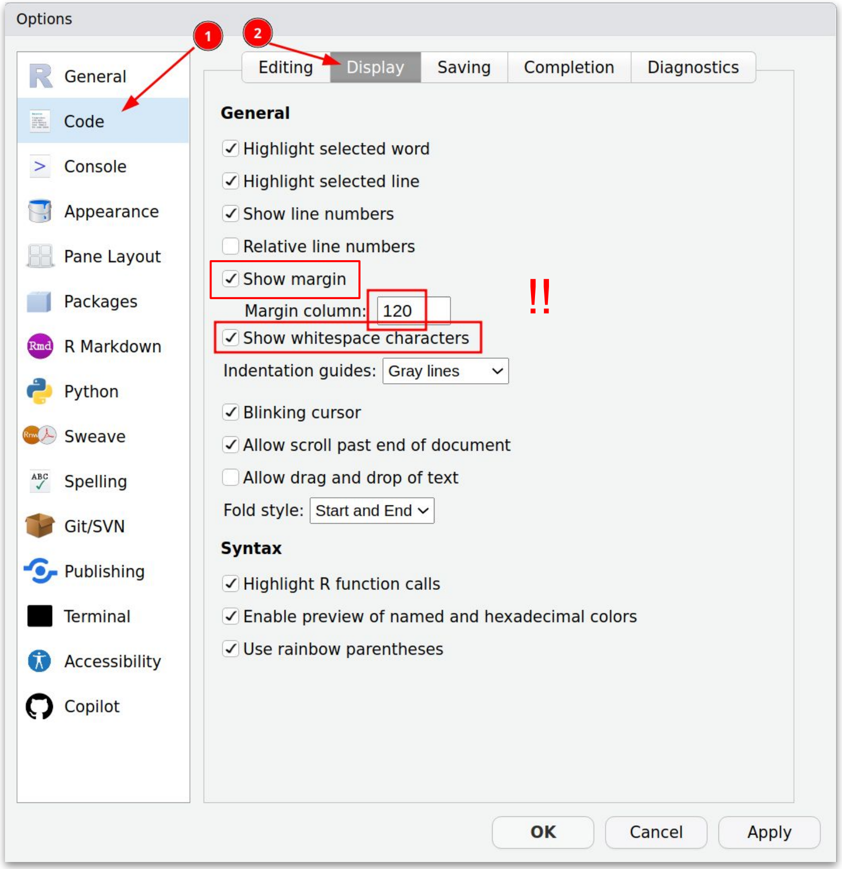
Empty newlines or trailing whitespaces are common problems. The following settings make these more visible.



Programming Style in ProgR

Recommended RStudio Settings

Empty newlines or trailing whitespaces are common problems. The following settings make these more visible.



General

- [✓] Highlight selected word
- [✓] Highlight selected line
- [✓] Show line numbers
- [ ] Relative line numbers
- [✓] Show margin

Margin column: 120

- [✓] Show whitespace characters

Indentation guides: Gray lines

- [✓] Blinking cursor
- [✓] Allow scroll past end of document
- [ ] Allow drag and drop of text

Fold style: Start and End

Syntax

- [✓] Highlight R function calls
- [✓] Enable preview of named and hexadecimal colors
- [✓] Use rainbow parentheses

What We Expect You to Know

ProgR Code Style

- Code formatting
  - Limit line length to 120 characters
  - Use spaces, not tabs for indentation (this is the default in RStudio)
  - Use spaces before curly braces, and use a newline after opening braces except when using 'else':
    ```
    if (x == 1)·{
      x.is.one <- TRUE
    }·else·{
      x.is.one <- FALSE
    }
    ```
  - Opening parentheses, curly braces and square brackets add two spaces of indentation each, except if they are opened **and** closed on the same line:
    ```
    vapply(1:10, function(x) {
    ····x^2
    ··},
    ··numeric(1)
    )
    ```
    but
    ```
    vapply(1:10, function(x) {
    ··x^2
    }, numeric(1))
    ```
  - Use spaces around most infix operators and on the outside (but not inside) of parentheses; exception: no spaces before function call parentheses. Space after a comma, but not before. Two spaces before comments that share a line with code.
    ```
    x·<-·(-1·+·1:2)·*·seq(from·=·2,·to·=·5)··# comment
    ```
  - Do not have any trailing whitespaces at the end of lines
- Code conventions
  - Use `lowerCamelCase` for functions, `dotted.case` for variables, and `UpperCamelCase` for classes.
  - Use double quotes (`"`), not single quotes (`'`) for strings
  - Use `TRUE` and `FALSE`, not `F` or `T`
  - Use `<-` for assignment, not `=` or `->` or `<<-`
  - Do not use `L` to denote integers
- Avoiding common bugs
  - Use `seq_len()` and `seq_along()` instead of `1:n`, `1:length(x)`
  - Do not use `== NA`, use `is.na()`
  - Do not use `c()` for single values: No `c(1)`, `c(x)`, only for `c(1, 2)` or `c(x, y, z)`
  - Use double-square-brackets to access individual vector elements: `x[[1]]`, not `x[1]`
  - Use `vapply()`, not `sapply()`
  - All function arguments following the third should be passed by name
- ProgR homework specifics
  - Do not load any libraries, even indirectly (no `library()`, `require()`, `::`, ...)

What We Expect You to Know

Programming Style

Make your program easy to read by:

1. Avoiding unnecessary noise and keeping the appearance of your code uniform.
2. Using all communication channels available to you, like variable / function names, comments, and idioms that your reader is familiar with.
3. Assuming your audience has tunnel vision and should be able to understand small blocks of your code at a time

(And remember, the code you hand in is checked for style automatically)