Applying functions across lists

Environments, data types, and classes

Lecture 7, CPSC 499 Fall 2018

`lapply` and `sapply`

- Applies a function to every item in a list
- `lapply` returns a list containing the output of the function for every item
- `sapply` returns a vector if possible
- Could use `sapply` with `length` function on the output of `gregexpr` from lab last week
- These functions also work on vectors, but there are not many cases where you need them on vectors

`mapply`

- Applies a function over multiple vectors or lists
- Each vector/list corresponds to one argument to the function
- Output is vector, array, or list, like output of `sapply`

tapply

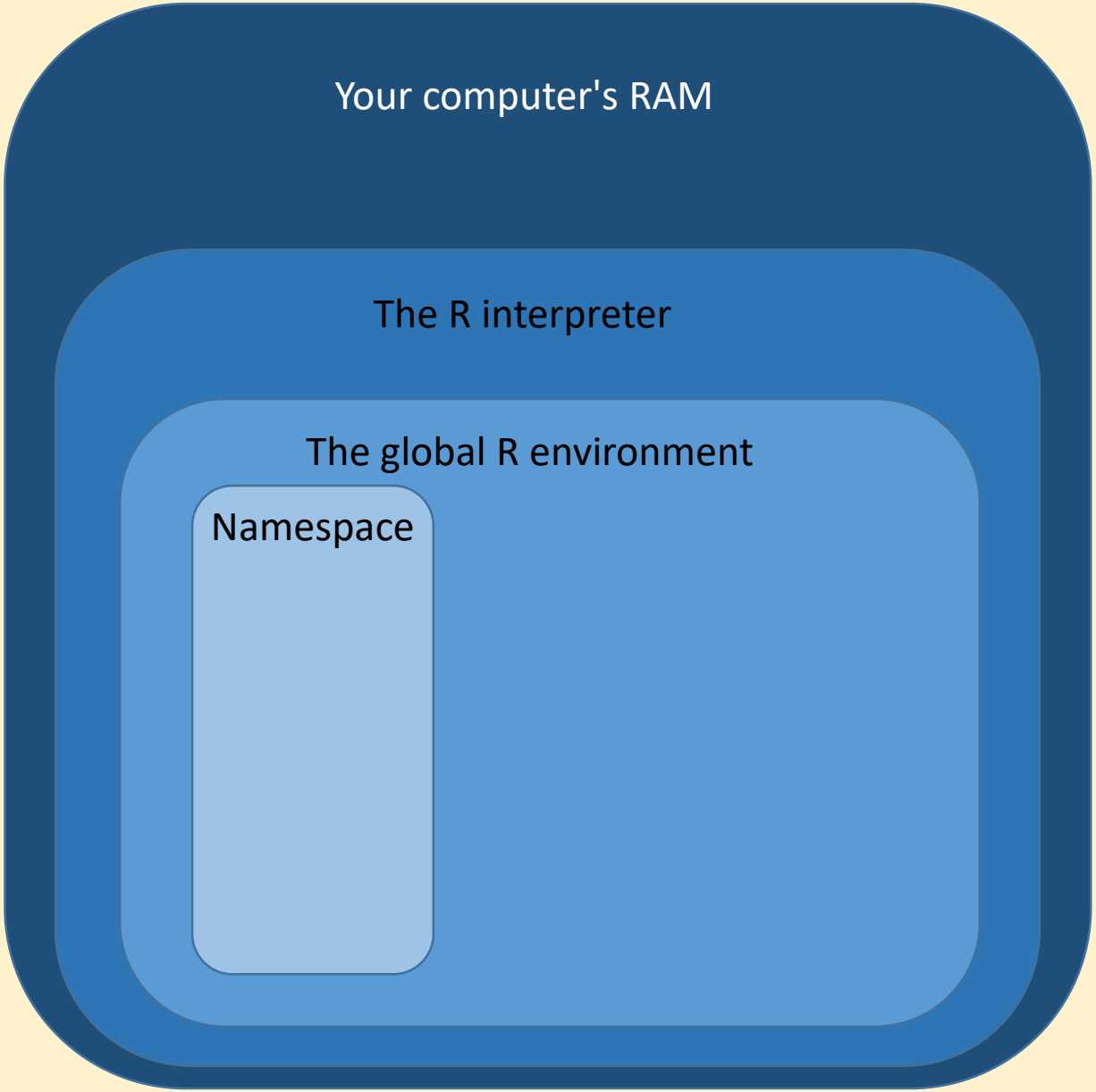
- We covered briefly in first week
- Split a vector up based on a grouping factor from another vector
- Apply a function to each sub-vector
- The `by` function can similarly split rows of a data frame up by a grouping factor

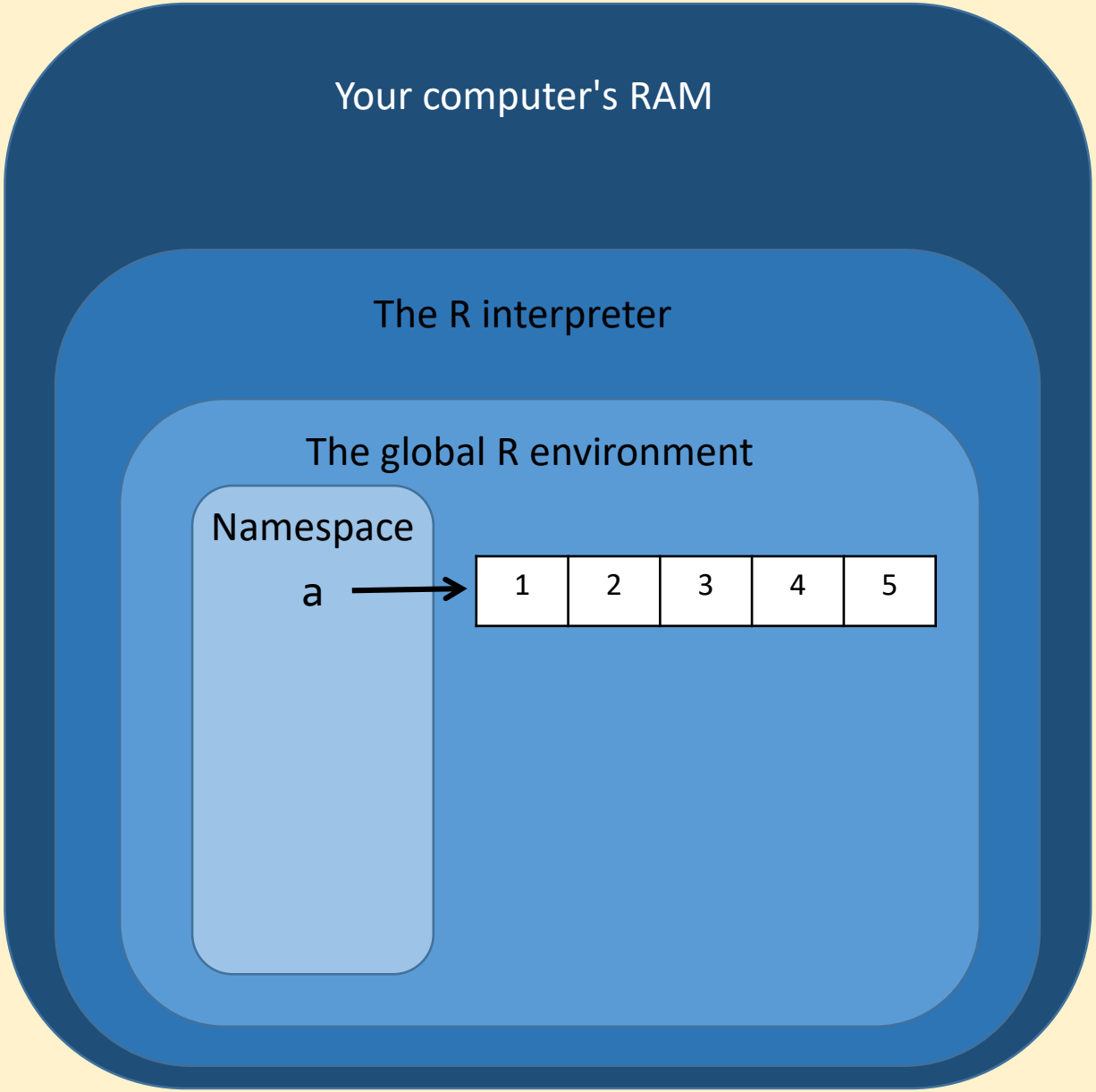
Mini exercise

- Take our SNP matrix
- Get genetic groups from Msi_groups_and_phenotypes.csv from Week 3
- Use `by` to get allele frequencies for each genetic group
- (Split the rows of the matrix by genetic group, then use colMeans on submatrices to get allele freq)

Environments and Namespaces

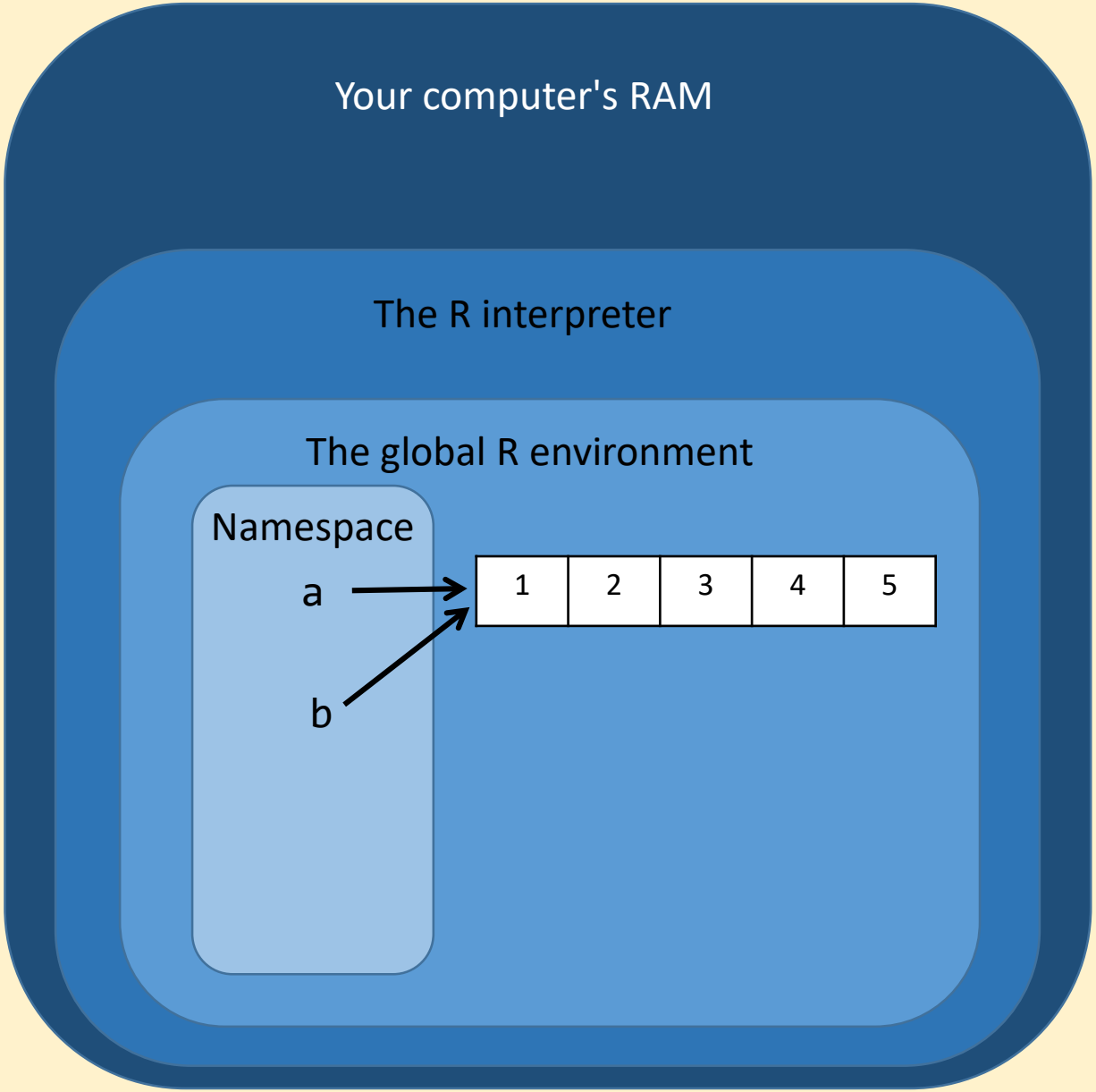
- The upper right corner of RStudio lists **objects** that are in your **environment**
- The **global environment** has user-created objects
- If you use the drop-down menu you can see **package environments**, which list functions available from R packages
- Every environment has a **namespace**, which is a list of all names that point to objects (including functions)


Your computer's RAM
The R interpreter
The global R environment
Namespace



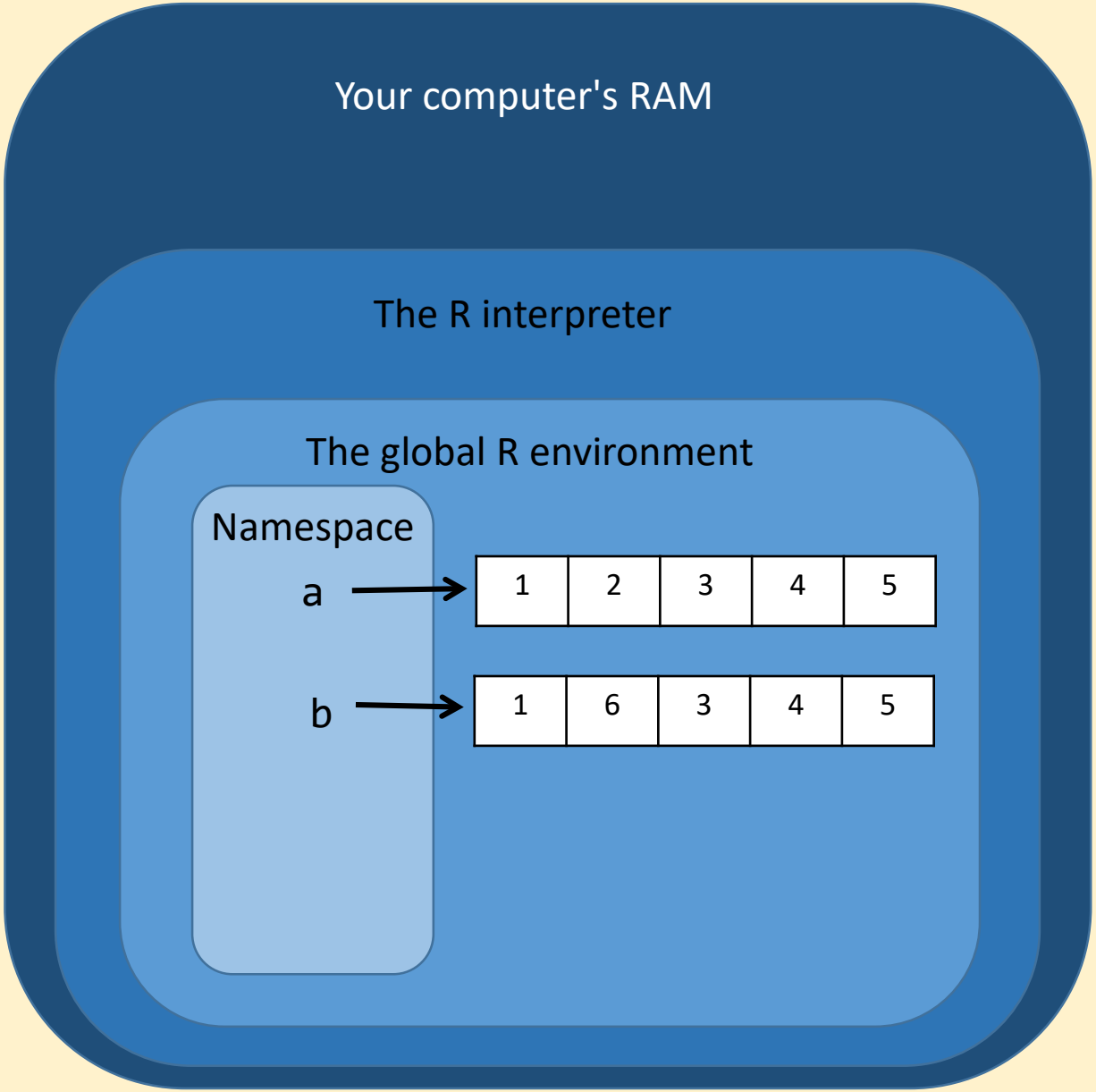
Commands you have typed:

```
a <- 1:5
```



Commands you have typed:

```
a <- 1:5
b <- a
```



Commands you have typed:

```
a <- 1:5
b <- a
b[2] <- 6
```

- Changing b didn't also change a. Instead, it made a copy and only changed b.
- This is called **copy-on-modify**.
- It's as though b and a were always different objects, except we didn't waste RAM on having two copies of the same thing.

Duplicate names across namespaces

- You can have the same name in multiple namespaces, referring to different objects in different environments
- R looks in the global environment first, then package environments, with most recently loaded package first
- :: can specify namespace, e.g. base::round

Functions have their own environments

- Objects you create or modify within the function environment don't exist/aren't modified in the global environment
- Functions can therefore reduce clutter in your global namespace
- Functions in your global environment DO however have access to the global namespace

Mini-exercise

- Try defining a function within the body of another function.
- When might you want to do this rather than defining both functions in the global environment?

Classes

- Every object in R belongs to one or more **classes**
- A class indicates what type of data is held in the object, and how that data can be displayed, manipulated, used, etc. in R
- We can use the `class` function to view the classes of an R object
- The `typeof` function gives related information about how the object is stored in memory

Common classes of vectors

- Two for numbers: **`numeric`** and **`integer`**.
  - We will cover these later in lecture.
- Character strings: **`character`**
- Booleans: **`logical`**
- Categorical variables: **`factor`**
  - Factors are an R-specific thing, useful for stats but less useful for conventional programming
- Dates: **`Date`**
- Date-time: **`POSIXct`** (number of seconds since beginning of 1970)

Functions for testing and converting vector classes

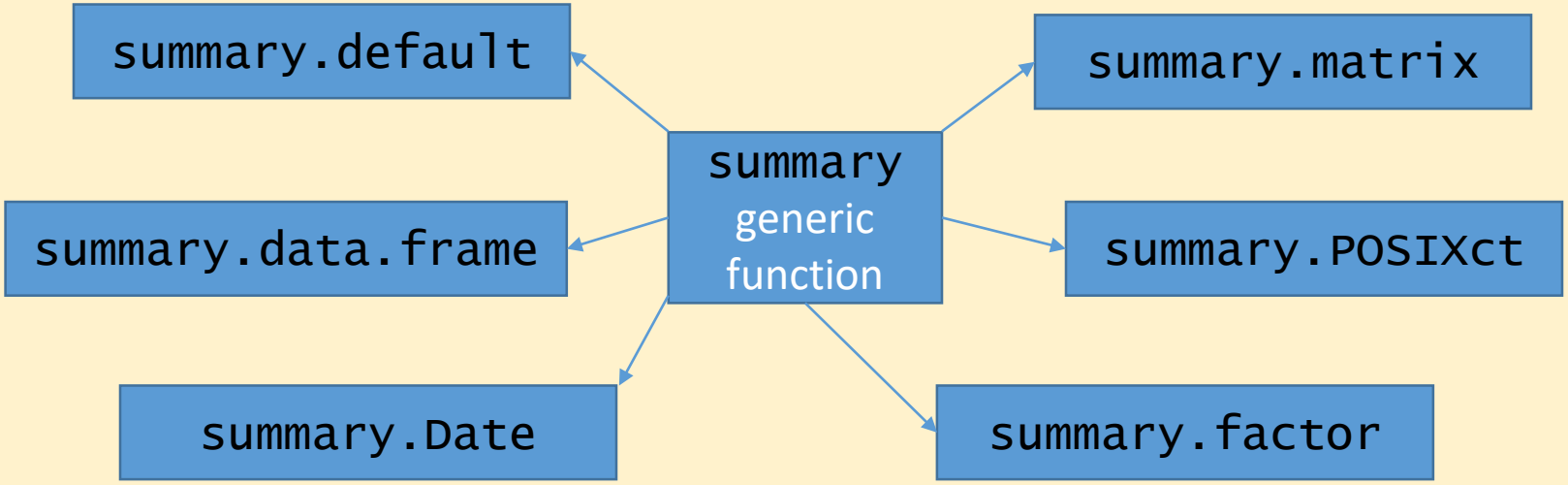
- Testing if a vector is a certain class: is.x
  - `is.numeric`, `is.integer`, `is.factor`, `is.character`
- Converting a vector from one class to another: as.x
  - `as.numeric`, `as.integer`, `as.factor`, `as.character`
- Although “vector” is not a class, there is an `is.vector` function

Functions for setting up vectors of specific classes

- `integer`, `numeric`, `character`, etc.
- These generally take one argument, which is the length of the vector
- Numbers: initializes filled with zeros
- Character: initializes filled with “” (empty string)
- Boolean: initializes filled with FALSE

Methods

- A **method** is just a function that is specific to a particular class
- **Object-oriented programming** is all about defining new classes and methods; we will learn more about this later.
- In R, a **generic function** is a function that tests the class of its first argument, then calls the appropriate method



Mini-exercise

- Type `methods(as.Date)` into the console
- What classes have `as.Date` methods?
- How do the arguments differ for those methods?

The ellipse (…) argument

- The last (or first) argument for any function definition can be `...`
- We see this a lot with generic functions and methods, but it can be used with any function
- Useful if there are a bunch of arguments that one function must pass internally to another
- Also used where any number of arguments can be provided (e.g. `c(...)`)

```
fun1 <- function(x, ...){
  y <- fun2(...)
}
```

Floating point numbers versus integers

So what is the difference between “integer” and “numeric”, anyway?

- We can make an integer vector and numeric vector that contain exactly the same numbers
- Using `object.size` we see that the integer vector takes up half as much RAM

Integer format

- In R an “integer” is always 32-bits (4 bytes)
- Can store numbers from −2,147,483,648 to 2,147,483,647
- In C this is called “long” integer (in contrast to a “short” 16-bit integer)
- This is why you can force a number to be an integer by putting an L on the end (e.g 10L)

Binary numbers

Decimal number	Binary number
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
2,000,000,000	1110111001101011001010000000000

- Basis for storing numbers in computer memory, and doing computations with them on the processor
- Any time a number is printed out on your screen, it has been converted from binary to decimal first
- Any time a number is read into memory from a file or your keyboard input, it is converted from the text string for a decimal number to the binary number
- One bit used for +/-

But weird things are happening with decimals…

- Last week we saw `colMeans` of our centered matrix were not exactly zero
- `0.3 - 0.2 - 0.1` is not zero
- We can use `print` with the `digits` argument to show that none of those three numbers is exactly what it is supposed to be

Mini exercise

- Experiment with different numbers, using print with digits = 18
- What numbers are represented exactly?
- Can you find a pattern?

Double-precision floating point numbers

- (Single-precision exists, just not in R)
- “numeric” class in R (also “double”)
- Total 64 bits (8 bytes) per number
  - 53 bits for an integer
  - 11 bits for an exponent
- In scientific notation we use exponents, e.g. $5.43 * 10^{-8}$
- Floating point numbers use base 2, e.g.

$7634 * 2^{-15}$

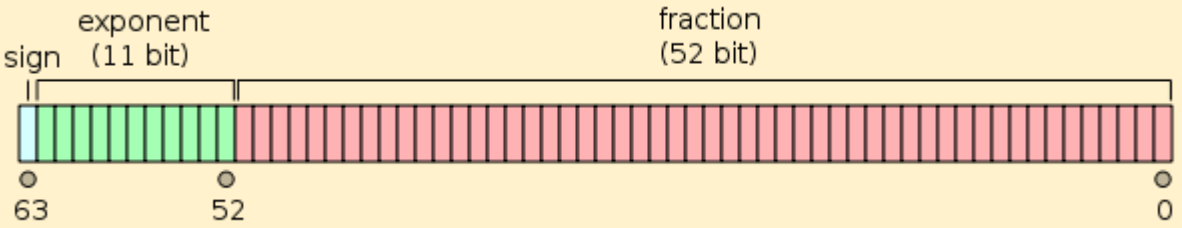


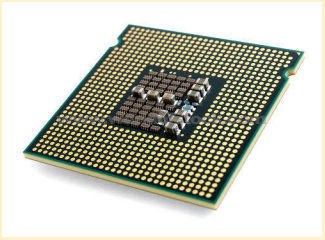
Image: Codekaizen, Wikipedia

The precision of double-precision floating point numbers

- Numbers that are represented exactly
  - All integers, i.e. anything * 2^0
  - Anything that is a multiple of a power of 2, for example 0.75 is the same as 3 * 2^{-2}
- All other numbers are precise to 16 significant digits or so
- In addition to very small numbers, can store very large numbers (> 10^{300}), just not precisely over 10^{16}
  - Compare to 32-bit integer going to ~10^9

Should you use “integer” or “numeric”?

- Unless you are working with thousands or millions of data points it really won’t matter
- If you know you will be working with integers (e.g. count data) it is better to use “integer”
  - Less RAM needed
  - Faster computation time
- Integer math and floating point math happen on physically different parts of your processor



How R decides between “integer” and “numeric”

- In `read.table`, any column that is purely integers is made “integer” by default
- If you make a series of numbers with the `:` operator, that will be “integer”
- Otherwise numbers typed into console are “numeric” by default
- `as.integer` or `as.numeric`
- `L` after number makes it an integer

Thursday’s lab

- We’ll look at integers vs. floating point numbers, and how to keep integers from getting converted
- You'll also see what it's like when a package adds classes to R