Introduction to R

Summer School

Alexey Fedoseev

June 3, 2025

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Interpreter versus Compiler

An *interpreter* translates high-level instructions into an intermediate form, which it then executes.

In contrast, a *compiler* translates high-level instructions directly into machine language.

Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long.

The interpreter, on the other hand, can immediately execute high-level programs.

R history

R has been around for a while, and is well-developed:

- Introduced in 1996 as an evolution of the S language.
- R is designed for exploring and analysing data.
- Home page: http://www.r-project.org.
- Community packages are stored at CRAN - Comprehensive R Archive Network.
- Bioinformatics packages are also stored at http://bioconductor.org.
- A new full version of R is released each year.

About R

R is used widely in a number of disciplines, like ecology, biology, etc., and provides a solid platform for beginner scientific programmers. Many plotting and statistical operations are built right into the language.

It is a broadly useful language. This means that you should be able to code most things using R, and it should be relatively easy to accomplish.

It runs on all major operating systems.

It is used by many scientific programmers and taught in many research science departments. This makes collaborating with experts on large or complex projects much easier.

Starting R

If you want to practise some commands during the session, you can install R from one of the following links:

For Windows: https://cran.r-project.org/bin/windows/base/

For macOS: https://cran.r-project.org/bin/macosx/

In order to start R open a terminal and type R if you have Linux or an Apple computer, or double-click on the R symbol if you have a Windows on the computer.

Let us know if you think it's not working. You are welcome to follow along by entering the commands on the slides, and playing with the output.

There are several graphical R interfaces available. These are handy, but we generally don't recommend them as in some cases they have serious drawbacks. However, you are welcome to use them if you like.

Version of R

After starting R the first thing you see is a version of R you are using.

```
R version 4.4.0 (2024-04-24) -- "Puppy Cup"
```

This is an important information because the developers of R constantly instroducing new features and if you write a program using the latest version of R and give it to someone who is using a very old version of R, it might not work due to the changes made in the language.

R prompt

After reading the greeting message you will find at the very bottom of it a greater sign ‘>’ followed by the cursor. Here you can type in your instructions. R executes them immediately after you hit Enter. Let us try to give our first instruction.

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> "hello"
[1] "hello"
>
```

As soon as the execution of instruction is finished, R displays the result of it and waits for new commands with a new prompt ‘>’.

Data types

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are just reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.

You may like to store information of various data types like integer (1,2,3, . . .) or real numbers (3.1415, 1.2, . . .) , logical values (TRUE, FALSE), etc.. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

In contrast to other programming languages like C and Java, in R the variables are not declared as some data type. The variables are assigned with a value and data type of this value becomes the data type of the variable.

Variables in R

Let us create our first variable. First you need to come up with a name of the variable that will describe what you are going to store in this variable.

If I want to store a number in a variable I will call it: `mynumber`. And the value I want to store is: 8. In order to “put” 8 in `mynumber` we use the *assignment* operator: `<-`.

After you finished typing the instruction, press `Enter`.

```
> mynumber <- 8
```

You read “`mynumber <- 8`” as “`mynumber` is assigned the value 8”.

Variables in R

Type the name of the variable and press `Enter` to see the value that is assigned to the variable.

```
> mynumber
[1] 8
```

The value of a variable could consist of several elements and the `[1]` indicates that the display starts at the first element of variable. We will have more examples later.

Note: The name of a variable in R must start with a letter (`A-Z` and `a-z`) and can include letters, digits (0–9), dots (.), and underscores (_). R discriminates between uppercase and lowercase letters in the name of the variable, so that `x` and `X` can name two distinct variables.

Data types

As we already know, the data type of the value determines the data type of the variable. So what is the data type of our variable `mynumber`? It is very easy to check.

```
> str(mynumber)
 num 8
```

We can see that R assigned a *numeric* type to our variable. It also showed us the value of this variable.

The command `str` displays the internal **str**ucture of an object in R

Numeric data type

We can check the data type of a number.

```
> str(2)
 num 2
> str(3.14)
 num 3.14
```

As you can see, numeric data type covers both integer and real numbers.

You can safely perform arithmetic operations within numeric data type.

```
> mynumber + 2
[1] 10
> mynumber - mynumber
[1] 0
```

Character data type

A character object is used to represent string values in R. A string is a collection of characters. It means we can put any message in a variable. All you have to do is to put it in quotes.

```
> mygreeting <- "Good Afternoon!"
> mymessage <- "It was a pleasure speaking with you today."
> mysignature <- "Sincerely, John."
> str(mygreeting)
 chr "Good Afternoon!"
```

A very common operation on strings is the operation of joining two strings together called *concatenation*.

Strings concatenation

Let us say you are writing an email. You have a variable `mygreeting` with a greeting phrase, a variable `mymessage` containing the body of your email and a variable `mysignature` holding you signature.

You can use the cat command to concatenate all your strings together in one message.

```
> cat(mygreeting, mymessage, mysignature, "\n")
Good Afternoon! It was a pleasure speaking with you today. Sincerely, John.
```

Symbol `\n` is a special character that signifies a **new line**. To fully understand its meaning try running this command without `\n` and see what happens.

Dynamic typing

You can re-use the same variable if you need to adjust the value. Let us say you want to change the greeting in our email.

```
> mygreeting <- "Good Morning!"
>
> cat(mygreeting, mymessage, mysignature, "\n")
Good Morning! It was a pleasure speaking with you today. Sincerely, John.
```

Notice that R allows you to re-assign a different data type to a variable that already has a value. R will overwrite the old value and it will be lost.

```
> str(mynumber)
 num 8
> mynumber <- "416-123-3456"
> str(mynumber)
 chr "416-123-3456"
```

Logical data type

Very often in programming we compare the values. For example:

```
> temperature <- 25
> temperature > 0
[1] TRUE
> temperature < 0
[1] FALSE
```

`TRUE` and `FALSE` are logical values and can be stored in a variable.

```
> is.cold <- (temperature < 0)
> str(is.cold)
 logi FALSE
```

Relational operators

Relational operators allow us to compare values.

```
> temperature
[1] 25
> temperature == 25
[1] TRUE
> temperature > 25
[1] FALSE
> temperature >= 25
[1] TRUE
> temperature <= 0
[1] FALSE
> temperature != 30
[1] TRUE
```

Logical operators

Often we need to combine the results of multiple comparisons. Logical operators are used to perform operations on logical values `TRUE` and `FALSE`.

Logical AND connects two "statements" that are eiter truthful or false. For example: if "it is sunny" AND "it is warm" then "I will go for a walk". What if "it is sunny" today, but it is -35°C? Then the statement "I will go for a walk" is false.

```
> temperature
[1] 25
> (temperature > 10) & (temperature < 28)
[1] TRUE
> (temperature > 10) & (temperature < 24)
[1] FALSE
> (temperature > -10) & (temperature < 0)
[1] FALSE
```

```
> TRUE & TRUE
[1] TRUE
> TRUE & FALSE
[1] FALSE
> FALSE & TRUE
[1] FALSE
> FALSE & FALSE
[1] FALSE
```

Logical operators

Logical OR helps you decide whether the combination of two “statements” is true or false.

```
> TRUE | TRUE
[1] TRUE
> TRUE | FALSE
[1] TRUE
> FALSE | FALSE
[1] FALSE
```

Logical NOT negates the logical value.

```
> is.cold
[1] FALSE
> !is.cold
[1] TRUE
```

Scripts

Interacting with R using the prompt is a convienient way to play with variables and data. However after you determined your instructions you should write them down separately from the R prompt. Why? Because after you close the prompt you can lose all your progress.

The set of instructions for the interpreted programming language (called the *source code*) saved in a plain text file is called the *script*.

To save the source code always use the plain text editor like Sublime Text, Atom, emacs, vi, nano, etc.. Do not use a heavy text processor like Microsoft Word! It adds a lot of characters to the file that the programming language does not understand.

Let us create our first script in R.

Scripts

Our script will be creating an email. Open the new tab in the text editor and type in the following code

```r
mygreeting <- "Good Afternoon!"
mymessage <- "It was a pleasure speaking with you today."
mysignature <- "Sincerely, John."

cat(mygreeting, mymessage, mysignature, "\n")
```

Save it as `email.R`. The extension `.R` indicates that is it an R script.

Scripts

Now you can run your script by runing the following command in your command line (`bash` for Linux and Apple computers, `git-bash` for Windows)

```
user@scinet scripts $ ls
email.R

user@scinet scripts $ Rscript email.R
Good Afternoon! It was a pleasure speaking with you today. Sincerely, John.
```

As you can see after running the script we are back in the terminal.

Comments

Automating your workflows with a programming language can be very fruitful. You realise it very fast and expand the functionality of your program. Rapidly it can contain hundreds and even thousands of line of code.

However, the common situation is that people tend to simply forget what they meant by a certain notation or calculation in the very beginning.

Using well named variables helps, but sometimes you just want to finish fast and did not give very descriptive names to your variables or put many commands together.

The best practice together with well named variables is to put comments in your code. It means that you put a small description or even your thoughts on what does this line do.

You can put a comment in your program using the # symbol. Everything after that symbol in the line will be treated as a comment and ignored. Comments are for people and not for computers!

```
> # This is a comment and would be ignored by the computer
> mynumber # This is a comment as well
[1] "416-123-3456"
```

Comments

Let us say that you have collected a phone number that consists of 3 variables and you want to display in a specific format.

```r
area.code <- "416" # 416 corresponds to an area code of Toronto
first.digits <- "123"
last.digits <- "3456"

# Merging the parts of a phone number together using "-" as a separator
# Example: 416-123-3456
cat(area.code, "-", first.digits, "-", last.digits, "\n", sep="")
```

Commenting your code is a very good practice. What seems obvious today could be obscure even a week later.

Ask yourself: will I remember what I have done here a year later?

Vectors

We know how to store a value in a variable. To view what value is stored in a variable we can simply type the name of the variable in R prompt.

```
> myapple <- "Big Red Delicious Apple"
> myapple
[1] "Big Red Delicious Apple"
```

Now we have a task to buy fruits and vegetables of the shopping list:

- Apple
- Orange
- Lemon
- Potato
- Cabbage

From what we already know, we can create a separate variable containing each item on our shopping list. However, there is a better way to store our list!

Vectors

We are going to use vectors to store our shopping list.

A vector is a sequence of elements of the same basic type.

In order to create a vector use the c command that combines values into a vector.

```
> fruits <- c("Apple", "Orange", "Lemon")
> str(fruits)
 chr [1:3] "Apple" "Orange" "Lemon"
> vegetables <- c("Potato", "Cabbage")
> str(vegetables)
 chr [1:2] "Potato" "Cabbage"
> shopping_list <- c(fruits, vegetables)
> str(shopping_list)
 chr [1:5] "Apple" "Orange" "Lemon" "Potato" "Cabbage"
```

Vectors

After we created the vector it is usually important to access its elements. Every element in a vector is referenced by the index. Adding the square brackets with the number inside to the variable name prints out the value stored in this variable at this index.

```
> fruits
[1] "Apple" "Orange" "Lemon"
> fruits[1]
[1] "Apple"
> fruits[2]
[1] "Orange"
> fruits[3]
[1] "Lemon"
> cat("I need to buy one", fruits[1], "\n")
I need to buy one Apple
```

Note that indices of the vector in R start with 1.

Vectors

We can access multiple elements in the vector by specifying the vector of indices. R provides us with a fast way to generate a vector of sequential numbers.

```
> fruits[c(1,3)]
[1] "Apple" "Lemon"
> 1:5
[1] 1 2 3 4 5
> fruits[1:2]
[1] "Apple" "Orange"
```

If we ask for the elements that are not in the vector, R will print out an `NA` value (Not Available/Missing Value).

```
> fruits[4]
[1] NA
> fruits[1:5]
[1] "Apple"  "Orange" "Lemon"  NA       NA
```

Vectors

When R encounters a negative index it skips the corresponding element. Notice that the `fruits` variable did not change.

```
> fruits
[1] "Apple"  "Orange" "Lemon"
> fruits[-1]
[1] "Orange" "Lemon"
> fruits
[1] "Apple"  "Orange" "Lemon"
```

Often it is useful to use logical values `TRUE` or `FALSE` to indicate which element to keep (`TRUE`) or remove (`FALSE`).

```
> fruits[c(TRUE, FALSE, TRUE)]
[1] "Apple" "Lemon"
```

Vectors

How many items are on my shopping list? Use the command `length` to count the number of elements in the vector.

```
> fruits
[1] "Apple"  "Orange" "Lemon"
> length(fruits)
[1] 3
> shopping_list
[1] "Apple"   "Orange"  "Lemon"   "Potato"  "Cabbage"
> length(shopping_list)
[1] 5
```

Do you have apples on your list? Check it using `%in%` command

```
> "Apple" %in% shopping_list
[1] TRUE
> "Mango" %in% shopping_list
[1] FALSE
```

Vectors

We can store numerical values and other data types in a vector as well.

```
> mynumbers <- c(1,5,3,7,5)
> str(mynumbers)
 num [1:5] 1 5 3 7 5
```

But, what happens if we create a vector of mixed data types?

```
> mymix <- c(FALSE, 1)
> str(mymix)
 num [1:2] 0 1
> mymix <- c(FALSE, 1, "Apple")
> str(mymix)
 chr [1:3] "FALSE" "1" "Apple"
```

R converts every element of the vector into the type that suits all values. At first, a boolean `FALSE` was converted in a numerical `0`. However, adding a string `"Apple"` converts every element into a string.

Vectors

We can easily perform operations on the whole vector.

```
> mynumbers
[1] 1 5 3 7 5
> mynumbers + 10
[1] 11 15 13 17 15
> mynumbers * mynumbers
[1]  1 25  9 49 25
```

Very often we want to establish what elements of the vector satisfy a particular condition. This is called conditional slicing or subsetting.

```
> (mynumbers > 2) & (mynumbers < 6)
[1] FALSE  TRUE  TRUE FALSE  TRUE
> mynumbers[(mynumbers > 2) & (mynumbers < 6)]
[1] 5 3 5
```

Vectors

To make our shopping list useful we need to add quantities. We want our shopping list to look like an Excel spreadsheet with rows and columns. Let us use already defined variable mynumbers as our quantities.

```
> mynumbers
[1] 1 5 3 7 5
> shopping.cart <- cbind(shopping_list, mynumbers)
> shopping.cart
     shopping_list mynumbers
[1,] "Apple"       "1"
[2,] "Orange"      "5"
[3,] "Lemon"       "3"
[4,] "Potato"      "7"
[5,] "Cabbage"     "5"
```

Notice how R converted all elements into strings. The resulting shopping.cart is represented as a **matrix**.

Matrices

A **matrix** is a collection of elements of the same data type arranged in a two-dimensional rectangular layout. You can think of a matrix as a mentioned earlier Excel spreadsheet.

Using the command cbind we can add more columns with details to our shopping cart.

```
> locations <- c("Costco", "Walmart", "Walmart", "Zehrs", "Walmart")
> shopping.cart <- cbind(shopping.cart, locations)
> shopping.cart
     shopping_list mynumbers locations
[1,] "Apple"       "1"       "Costco"
[2,] "Orange"      "5"       "Walmart"
[3,] "Lemon"       "3"       "Walmart"
[4,] "Potato"      "7"       "Zehrs"
[5,] "Cabbage"     "5"       "Walmart"
```

Column names

We can improve our `shopping.cart` by renaming our columns using `colnames` command.

```
> colnames(shopping.cart)
[1] "shopping_list" "mynumbers" "locations"
> colnames(shopping.cart) <- c("Items", "Quantities", "Locations")
> shopping.cart
     Items     Quantities Locations
[1,] "Apple"   "1"        "Costco"
[2,] "Orange"  "5"        "Walmart"
[3,] "Lemon"   "3"        "Walmart"
[4,] "Potato"  "7"        "Zehrs"
[5,] "Cabbage" "5"        "Walmart"
```

Data frames

Our shopping cart looks good, however quantities should be numbers instead of strings. It is important as we can perform mathematical operations on numbers and not on strings. *Data frames* allow us to store various data types in one variable. Since data frame type is more suitable for our shopping cart, we will overwrite the variable `shopping.cart`.

```
> shopping.cart <- data.frame(shopping_list, mynumbers, locations)
> shopping.cart
  shopping_list mynumbers locations
1         Apple         1    Costco
2        Orange         5   Walmart
3         Lemon         3   Walmart
4        Potato         7     Zehrs
5       Cabbage         5   Walmart
```

How to choose between a matrix or a data frame? You should use matricies if you perform a lot of mathematical operations on a very large amount of numbers. This will ensure your calculations to be more efficient. Otherwise, use data frames.

Data frames

Renaming the columns in a data frame as easy as in a matrix.

```
> colnames(shopping.cart) <- c("Items", "Quantities", "Locations")
> shopping.cart
    Items Quantities Locations
1   Apple          1    Costco
2  Orange          5   Walmart
3   Lemon          3   Walmart
4  Potato          7     Zehrs
5 Cabbage          5   Walmart
```

Data frames

We can verify that our quantities in fact are numbers by using the command `str`.

```
> str(shopping.cart)
'data.frame':   5 obs. of  3 variables:
 $ Items      : Factor w/ 5 levels "Apple","Cabbage",..: 1 4 3 5 2
 $ Quantities: num  1 5 3 7 5
 $ Locations : Factor w/ 3 levels "Costco","Walmart",..: 1 2 2 3 2
```

Notice that Items and Locations have a factor data type which we have not seen before.

Note: in newer versions on R you need to add stringsAsFactors=TRUE to the list of parameters for the command `data.frame` to get the same output, as the default behavior of this command has changed. For example:

```
shopping.cart <- data.frame(shopping_list, mynumbers,
                            locations, stringsAsFactors=TRUE)
```

Factors

Factors help us to find out the categories in a vector and how are they all doing.

Since we only shop in Costco, Walmart and Zehrs, they will be our categories or `levels`.

```
> locations
[1] "Costco"  "Walmart" "Walmart" "Zehrs"   "Walmart"
> places <- factor(locations)
> places
[1] Costco  Walmart Walmart Zehrs   Walmart
Levels: Costco Walmart Zehrs
> summary(places)
 Costco Walmart   Zehrs
      1       3       1
```

The command `summary` allows us to see how often each level appears in the vector.

summary

The `summary` command actually gives us more information if we use it on our `shopping.cart`.

```
> summary(shopping.cart)
     Items     Quantities    Locations
 Apple   :1   Min.   :1.0   Costco :1
 Cabbage:1   1st Qu.:3.0   Walmart:3
 Lemon   :1   Median :5.0   Zehrs  :1
 Orange :1   Mean   :4.2
 Potato :1   3rd Qu.:5.0
             Max.   :7.0
```

The column Quantities shows us the statistics that are useful for the data analysis. For the columns with factors `summary` gives us the frequencies of the levels.

Selecting columns

There are three ways to select a whole column in a data frame that are all equivalent to each other. Additionally, you can refer to the column by its index (which is less descriptive but useful sometimes).

```
> shopping.cart[["Quantities"]]
[1] 1 5 3 7 5
> shopping.cart[,"Quantities"]
[1] 1 5 3 7 5
> shopping.cart$Quantities
[1] 1 5 3 7 5
> shopping.cart[,2]
[1] 1 5 3 7 5
```

If you are using the dollar sign $ to extract the actual column and the column name has special characters, it **must** be surrounded by quotes (`shopping.cart$"Price/kg"`) or backticks (``shopping.cart$`Price/kg` ``).

Adding columns to a data frame

There is a convinient way to add a column to the data frame.

```
> prices.kg <- c(3.27, 3.9, 3.97, 3.49, 2.14)
> shopping.cart$Price <- prices.kg
> colnames(shopping.cart)
[1] "Items"      "Quantities" "Locations"  "Price"
> colnames(shopping.cart)[4] <- "Price/kg"
> shopping.cart
    Items Quantities Locations Price/kg
1   Apple          1    Costco     3.27
2  Orange          5   Walmart     3.90
3   Lemon          3   Walmart     3.97
4  Potato          7     Zehrs     3.49
5 Cabbage          5   Walmart     2.14
```

Selecting rows

You can extract rows from the data frame using slicing.

```
> shopping.cart[c(2,4),]
   Items Quantities Locations Price/kg
2 Orange          5   Walmart     3.90
4 Potato          7     Zehrs     3.49
> shopping.cart[c(2,4), c("Items", "Price/kg")]
   Items Price/kg
2 Orange     3.90
4 Potato     3.49
> shopping.cart[2,"Items"]
[1] Orange
Levels: Apple Cabbage Lemon Orange Potato
```

Notice how selecting rows preserves all levels for the factors.

Conditional slicing or subsetting data

While looking at your shopping list you want to know if you need to bring the car and what items you are buying at Walmart.

```
> shopping.cart$Quantities > 4
[1] FALSE  TRUE FALSE  TRUE  TRUE
> shopping.cart[shopping.cart$Quantities > 4,]
    Items Quantities Locations Price/kg
2  Orange          5   Walmart     3.90
4  Potato          7     Zehrs     3.49
5 Cabbage          5   Walmart     2.14
> shopping.cart$Location == "Walmart"
[1] FALSE  TRUE  TRUE FALSE  TRUE
> shopping.cart[shopping.cart$Location == "Walmart",]
    Items Quantities Locations Price/kg
2  Orange          5   Walmart     3.90
3   Lemon          3   Walmart     3.97
5 Cabbage          5   Walmart     2.14
```

Adding columns

You are also curious how much money you need to take with you to the store.

```
> shopping.cart$Total <- shopping.cart$Quantities * shopping.cart$`Price/kg`
> shopping.cart
    Items Quantities Locations Price/kg Total
1   Apple          1    Costco     3.27  3.27
2  Orange          5   Walmart     3.90 19.50
3   Lemon          3   Walmart     3.97 11.91
4  Potato          7     Zehrs     3.49 24.43
5 Cabbage          5   Walmart     2.14 10.70
```

Using R command sum we can easily calculate how much money we need to have on us.

```
> cat("Total amount:", sum(shopping.cart$Total), "\n")
Total amount: 69.81
```

Loops

Repetitive tasks

Computers are very good at repeating commands!

You want to congratulate your friends, but you have so many of them!

```
> cat("Happy Birthday Erik!\n")
Happy Birthday Erik!
> cat("Happy Birthday John!\n")
Happy Birthday John!
> cat("Happy Birthday Alice!\n")
Happy Birthday Alice!
> cat("Happy Birthday Mike!\n")
Happy Birthday Mike!
```

These are only 4 friends out of hundreds that you have. If you want to change your congratulation message you would rather *not* change it hundred times.

Loops

Loops give you control over repetitive tasks. Loop means: ‘for’ each friend name ‘in’ friends vector repeat the commands inside the *code block* surrounded by the curly braces {}.

```
> friends <- c("Erik", "John", "Alice", "Mike")
> for (friend_name in friends) {
+   cat("Happy Birthday ", friend_name, '!\n', sep = "")
+ }
Happy Birthday Erik!
Happy Birthday John!
Happy Birthday Alice!
Happy Birthday Mike!
```

To change your congratulation message now you need to modify one line only! The additional parameter `sep = ""` for the command `cat` indicates that we want to concatenate our strings using *separator* empty string (`""`). It means that R with paste an empty string `""` between every string in our command. Necessarily we need to provide spaces at the ends of our strings ourselves.

Conditional statements

What if you have a very special friend? ‘If’ it is a special friend be specific, ‘else’ be general.

```
> friends <- c("Erik", "John", "Alice", "Mike")
> for (friend_name in friends) {
+   cat("Happy Birthday ", friend_name, '! ', sep = "")
+   if (friend_name == "Erik") {
+     cat("I wish you to stay healthy and feel great!\n")
+   }
+   else
+     cat("Best wishes!\n")
+ }
Happy Birthday Erik! I wish you to stay healthy and feel great!
Happy Birthday John! Best wishes!
Happy Birthday Alice! Best wishes!
Happy Birthday Mike! Best wishes!
```

You do not have to specify curly braces {} if your *code block* consists of only one command.

Loops

You want to wish your friends something amazing! Create another vector with your wishes and use the same index to go through both vectors.

```
> friends <- c("Erik",       "John",       "Alice",         "Mike")
> wishes <- c("feel great", "stay happy", "feel beautiful", "stay healthy")
> friends.indices <- seq_along(friends) # same as 1:length(friends)
> friends.indices
[1] 1 2 3 4
> for (index in friends.indices) { # for every 'element' in a 'vector'
+   cat("Happy Birthday ", friends[index], '! Wishing you to ',
+       wishes[index], '!\n', sep = "")
+ }
Happy Birthday Erik! Wishing you to feel great!
Happy Birthday John! Wishing you to stay happy!
Happy Birthday Alice! Wishing you to feel beautiful!
Happy Birthday Mike! Wishing you to stay healthy!
```

if...else statement

```
> ingredients <- c("sugar", "milk", "water", "flour", "salt", "vanilla")
> for (ingredient in ingredients) {
+   if (ingredient %in% c("sugar", "flour", "salt"))
+     cat(ingredient, "is measured in tablespoons\n")
+   else if (ingredient %in% c("water", "milk"))
+          cat(ingredient, "is measured in cups\n")
+        else if (ingredient %in% c("apple", "banana"))
+               cat(ingredient, "is measured in pounds (lb)\n")
+             else cat(ingredient, "is measured in gramms\n")
+ }
sugar is measured in tablespoons
milk is measured in cups
water is measured in cups
flour is measured in tablespoons
salt is measured in tablespoons
vanilla is measured in gramms
```

User input

What if you want to have more friends? You can create one.

```
> your_name <- readline(prompt = "Enter your name: ")
Enter your name: Alice
> cat("Hi ", your_name, '!\n', sep="")
Hi Alice!
```

Command `readline` allows you to ‘interact’ with R prompt, meaning you can type in your name and press `Enter` to let R know that you finished typing.

Conditional loop

'While' loop can repeat the task `while` certain condition is `TRUE`. Change of the condition to `FALSE` ends the loop. Use it when you do not how many times the task needs to be performed.

```
> user_input <- ""
> while (user_input != "stop") {
+   user_input <- readline(prompt = "\nWhat should I do? ")
+   cat(your_name, " wants me to ", user_input, ".", sep="")
+   if (user_input == "stop") cat(" Bye.\n")
+ }

What should I do? walk
Alice wants me to walk.
What should I do? dance
Alice wants me to dance.
What should I do? stop
Alice wants me to stop. Bye.
```

Functions

Recipe

Today we are making a raisin bread! It is simple

```
> cat("Whip the butter\n")
Whip the butter
> cat("Add sugar\n")
Add sugar
> cat("Add flour\n")
Add flour
> cat("Add raisins\n")
Add raisins
> cat("Mix everything well\n")
Mix everything well
> cat("Bake at 150C for 1 hour\n")
Bake at 150C for 1 hour
```

What if you want to bake one more cake? Let's automate it!

Defining a function

We can group all these steps into a procedure of making a raisin bread.

```
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+   cat("Add sugar\n")
+   cat("Add flour\n")
+   cat("Add raisins\n")
+   cat("Mix everything well\n")
+   cat("Bake at 150C for 1 hour\n")
+ }
```

This is called a function. You simply combine several steps together and give it a distinctive name.

Note! The body of your function begins and ends with these symbols: `{ }` (knows as curly braces).

Calling a function

Next time you bake a new cake just call the function!

- If you ever do something more than once in your code, make a function to do it.
- By utilizing functions you can easily change any step within your function, for example if you want to use "brown sugar" instead of "sugar" you only change it in a function once.
- I cannot stress enough how important functions are.
- Your code will be much easier to read.
- Your functions could be used in your other programs.
- **Remember**: function names cannot start with a number.

```
> cat("I am making two cakes.\n")
I am making two cakes.
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
```

Calling a function

If you can't remember what the function is, you can just type its name (without parenthesis) and its definition will be displayed in your R terminal.

```
> make.raisin.bread
function() {
  cat("Whip the butter\n")
  cat("Add sugar\n")
  cat("Add flour\n")
  cat("Add raisins\n")
  cat("Mix everything well\n")
  cat("Bake at 150C for 1 hour\n")
}
```

Structuring your code

You will note that this recipe requires a lot of mixing. We can put these steps in a function as well!

```
> mix.ingredients <- function() {
+   cat("Add sugar\n")
+   cat("Add flour\n")
+   cat("Add raisins\n")
+   cat("Mix everything well\n")
+ }
```

Now our baking procedure can use this function

```
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+   mix.ingredients()
+   cat("Bake at 150C for 1 hour\n")
+ }
```

Arguments

What if we want to reuse the mixing function with other ingredients? You can specify what exactly you want to mix and in what order.

```
> mix.ingredients <- function(ingredient1, ingredient2, ingredient3) {
+   cat("Add ", ingredient1, "\n", sep="")
+   cat("Add ", ingredient2, "\n", sep="")
+   cat("Add ", ingredient3, "\n", sep="")
+   cat("Mix everything well\n")
+ }
> mix.ingredients("brown sugar", "flour", "raisins")
Add brown sugar
Add flour
Add raisins
Mix everything well
```

In programming the parameters of a function (in our example named `ingredient1`, `ingredient2`, `ingredient3`) are called **arguments of a function**.

Arguments

```
> mix.ingredients("chicken", "lettuce", "croutons")
Add chicken
Add lettuce
Add croutons
Mix everything well
```

R expects that you will provide exactly 3 arguments for the function `mix.ingredients`, unless you specify default values for certain arguments, which we will discuss later today.

```
> mix.ingredients("milk")
Add milk
Error in cat("Add ", ingredient2, "\n", sep = "") :
  argument "ingredient2" is missing, with no default
```

Arguments

Perhaps you need to mix 2 or 5 ingredients? Vectors can help us in this situation. A vector is a data structure that contains multiple elements of the same type.

```
> ingredients <- c("sugar","flour","raisins","vanilla","a pinch of salt")
> for (ing.index in seq_along(ingredients))
+   cat("Argument no.", ing.index, "is", ingredients[ing.index], "\n")
Argument no. 1 is sugar
Argument no. 2 is flour
Argument no. 3 is raisins
Argument no. 4 is vanilla
Argument no. 5 is a pinch of salt
```

Note: elements of a vector have to be of the **same type**. If your arguments have different types you need to use lists instead of vectors.

Arguments

If the number of parameters that your function needs can vary, define one argument as a vector containing as many ingredients as we need and work with each element of this vector as an argument.

```
> mix.ingredients <- function(ingredients) {
+   for (ingredient in ingredients)
+     cat("Add", ingredient, "\n")
+   cat("Mix everything well\n")
+ }
> ingredients <- c("sugar","flour","raisins","vanilla","a pinch of salt")
> mix.ingredients(ingredients)
Add sugar
Add flour
Add raisins
Add vanilla
Add a pinch of salt
Mix everything well
```

Local variables

Let us use our advanced mixing procedure to bake another cake.

```
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+
+   myingredients <- c("sugar","flour","raisins","vanilla","a pinch of salt")
+   mix.ingredients(myingredients)
+
+   cat("Bake at 150C for 1 hour\n")
+ }
```

`myingredients` referenced above is a variable and is only seen inside of the function. It is not visible outside of the function, it is a ‘local variable’. Let us follow the example below.

Local variables

If you try to use a variable, that is local to a function somewhere outside of this function, R will throw an error.

```
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Add vanilla
Add a pinch of salt
Mix everything well
Bake at 150C for 1 hour
> cat(myingredients)
Error in cat(myingredients) : object 'myingredients' not found
```

Global variables

Variables which are declared outside of functions are called ‘global variables’. You can have a variable outside of your function with the same name. However, it is advisable to come up with a different name for your variable, simply because you will not confuse them accidently.

```
> myingredients <- c("potatoes", "carrots", "peas")
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Add vanilla
Add a pinch of salt
Mix everything well
Bake at 150C for 1 hour
> cat(myingredients, "\n")
potatoes carrots peas
```

Global variables

You cannot rely on the variables outside of the function. Why?

```
> ingredients <- c("sugar", "flour", "raisins")
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+
+   mix.ingredients(ingredients)
+
+   cat("Bake at 150C for 1 hour\n")
+ }
```

This code will work. However, what happens if you redefine your ingredients for another recipe?

Global variables vs Local variables

```
> ingredients <- c("onions", "garlic", "tomatoes", "herbs")
> make.raisin.bread()
Whip the butter
Add onions
Add garlic
Add tomatoes
Add herbs
Mix everything well
Bake at 150C for 1 hour
```

Oh no! It is not a raisin bread anymore! This could be a source of an error that is **really hard to trace**. Always specify any external to the function variables as arguments!

Use arguments instead of global variables

At some point you will modify your global variable. It will affect functions that rely on this global variable. A better alternative is to use arguments of the function to specify anything you need outside of that particular function to make your codes reliable.

```
> make.raisin.bread <- function(ingredients) {
+   cat("Whip the butter\n")
+   mix.ingredients(ingredients)
+   cat("Bake at 150C for 1 hour\n")
+ }
> ingredients <- c("sugar", "flour", "raisins")
> make.raisin.bread(ingredients)
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
```

return statement

The result of the your function can be re-used later. For this purpose we need to tell what symbolizes the result of the function.

```
> make.raisin.bread <- function(ingredients) {
+   cat("Whip the butter\n")
+   mix.ingredients(ingredients)
+   cat("Bake at 150C for 1 hour\n")
+   ingredients.string <- paste(ingredients, collapse=", ")
+   result <- paste("Raisin bread made out of", ingredients.string)
+   return(result)
+ }
```

The function `paste` concatenates ingredients in one string. In order to add commas between the words we use the `collapse=", "` argument.

```
> paste(c("strawberry", "banana"), collapse=", ")
[1] "strawberry, banana"
```

return statement

```
> ingredients <- c("sugar", "flour", "raisins")
> my.bread <- make.raisin.bread(ingredients)
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
> cat("What did you make today?\n")
What did you make today?
> cat(my.bread, "\n")
Raisin bread made out of sugar, flour, raisins
```

Default arguments

Now we want to make a bolognese sauce. We can serve it with different types of pasta, however it goes best with spaghetti. If we know the best value for our argument we can specify a default value.

```
> make.pasta.bolognese <- function(sauce.ingredients, pasta = "spaghetti") {
+   ingredients.string <- paste(sauce.ingredients, collapse=", ")
+   cat("Cook", ingredients.string, "together\n")
+   cat("Serve on top of", pasta, "\n")
+   return(paste("Bolognese sauce with", pasta))
+ }
> make.pasta.bolognese(c("onions", "ground beef", "tomatoes"))
Cook onions, ground beef, tomatoes together
Serve on top of spaghetti
[1] "Bolognese sauce with spaghetti"
```

Notice we did not specify the pasta argument, so by default it will use “spaghetti”.

Default arguments

Do you only have fettuccine? It will work as well

```
> make.pasta.bolognese(c("onions", "ground beef", "tomatoes"), "fettuccine")
Cook onions, ground beef, tomatoes together
Serve on top of fettuccine
[1] "Bolognese sauce with fettuccine"
```

Additionaly, you can specify the name and a value of an argument which makes your code well documented.

```
> make.pasta.bolognese(
+   sauce.ingredients = c("onions", "ground beef", "tomatoes", "herbs"),
+   pasta = "spaghettini")
Cook onions, ground beef, tomatoes, herbs together
Serve on top of spaghettini
[1] "Bolognese sauce with spaghettini"
```

Use scripts to save you work

When you exit R prompt, you have an option to save your workspace. It means that all your variables and functions will be available next time you open R prompt. However if you do not save your workspace, all your work will be lost.

If you continue using your workspace to save your work you expect certain functions and variables to be available all the time. It is easy to forget that a month ago you defined an important variable that you keep using. So you submit your code without defining this variable assuming others have it as well. But we don't! Consequently your code does not work on other computers and R throws an error `object not found`.

Writing your code in scripts preserves your work and allows to reuse your code on other computers.

High-performance R

High-performance R

Just a reminder:

- R is an interpreted language. As such, there is an extra layer of infrastructure (the interpreter) needed to make R run
- As a general rule, because of the extra layer of infrastructure, interpreted languages (R, Python, Bash, Perl, . . .) are not high-performance languages
- True high-performance languages are compiled, and thus they lack the extra layer of infrastructure: C, C++, Fortran
- That being said, there are ways of making R better.

R and memory

One must be cognisant of how R manages memory:

- R is “pass by value” if the variables being passed are being modified within the function. As such, R frequently needs to make temporary copies of variables, and hitting the memory limit of your machine can be a frequent problem
- Like many dynamic languages, R relies on “garbage collection” to limit its memory usage
- In a running code, ”every so often” a garbage collection task runs and deletes variables that won’t be used any more
- You can force the garbage collector to run at any given time by calling `gc()`, but this almost never fixes anything significant
- How can GC know that you’re not going to use that big variable in the next line? The garbage collector needs your help to be effective

Useful memory-management commands

- `gc(verbose = TRUE)`, or just `gc(TRUE)`
  - Calling `gc(TRUE)` alone probably won't help anything, but it does give verbose output, returning memory usage as a matrix
- `ls()`
  - Lists all existing variables, as strings
- `object.size(variablename)`
  - Pass it a variable, and it prints out its size
  - Pass it get("variablename") and it will also print its size
- `rm(variablename)`
  - Deletes a variable you no longer need. Lets gc go to work
- Fun little one-liner which prints out all variables by size in bytes

```
> sort(sapply(ls(), function(x) {object.size(get(x))}), decreasing = TRUE)
```

object.size and gc

```
> gc()
          used  (Mb) gc trigger  (Mb)  max used   (Mb)
Ncells  183250   9.8     407500  21.8    350000   18.7
Vcells  377223   2.9     905753   7.0    864975    6.6
> get.mem <- function() return(gc()[, 1:2])
> old.mem <- get.mem()
> x <- rep(0., (16 * 1024)**2)
> xsize <- object.size(x)
> xsize
2147483688 bytes
> print(xsize, units = "MB")
2048 Mb
> get.mem() - old.mem
             used   (Mb)
Ncells        445      0
Vcells  268436139   2048
```

object.size and gc, some more

Now let's delete the object and see how the system memory behaves

```
> rm(x)
> final.mem <- get.mem()
> final.mem - old.mem
         used  (Mb)
Ncells    451   0.1
Vcells   1781   0.0
```

Be sure to delete temporary variables in your scripts, especially large ones!

Profiling

To push your code to new heights of awesome, or to make it useful at all (depending on your situation), you will need to profile your code. What is profiling?

- Profiling is analyzing where the code is spending its time. Which parts of the code are slowest?
- Testing how long individual functions take can be performed with the 'microbenchmark' package, or more crudely, `system.time`
- To test the whole program we use the 'Rprof' package

We'll do some examples of each.

Profiling individual functions

- The `system.time` command uses the OS's `time` command to determine how long the code takes to run.

```
> f <- function() {
+   a <- 1
+   for (i in 1:1e8) {
+     a <- a + i
+   }
+ }
> system.time(f())
   user  system elapsed
  0.964   0.002   0.967
```

Profiling individual functions - `microbenchmark`

- The `microbenchmark` function is more systematic. It takes an average over 100 calls of the function. Consequently, it can take a while to run
- Note that the microbenchmark package will need to be downloaded

```
> microbenchmark(f(), times=10)
Unit: milliseconds
 expr     min       lq     mean   median       uq      max neval
  f() 927.771 929.782 939.4129 940.1643 943.3133 952.1787    10
```

Profiling individual functions - `microbenchmark`

- You can use `microbenchmark` to compare performance of multiple functions:

```
x = runif(100)
microbenchmark(
  sqrt(x),
  x ^ 0.5
)
```

```
Unit: nanoseconds
    expr  min   lq    mean median     uq   max neval
 sqrt(x)  287  328  604.34    369  430.5 10742   100
   x^0.5 1968 2009 2106.17   2050 2091.0  5248   100
```

Profiling whole programs

Use Rprof to analyse where the code is spending its time.

```
> addme <- function(a, b) { Sys.sleep(0.001); return(a + b) }
> test <- function() {
+   a <- 1
+   for (i in 1:1e5)
+     a <- addme(a, i)
+ }
> Rprof("Rprof.data")
> test()
> Rprof(NULL)
> s <- summaryRprof("Rprof.data")
> s$by.total
                            total.time total.pct self.time self.pct
"test"                            1.82     100.0      0.00      0.0
"Sys.sleep"                       1.80      98.9      1.80     98.9
"addme"                           1.80      98.9      0.00      0.0
```

Rprof

Some notes about the last slide:

- `Rprof` samples the program every 20ms, by default, to see where the program is spending its time
- Use `Rprof("filename")` to store the `Rprof` results in a particular file
- Use `Rprof(NULL)` to turn off profiling
- You can read "filename" if you want. It's easier to just use `summaryRprof("filename")` to analyse the results
- Results are given in data frames
- Columns `total.time` and `total.pct` (total percent) include all time spent within a function, including calls to other functions
- `self.time` and `self.pct` indicate actually real time spent in each function (`self.pct` should add up to 100%, give or take rounding).

foreach and doparallel

All modern computers have many processors, and to take advantage of this power, we need to switch from serial code to a parallel version. If your code already uses functions or libraries that leverage multiple processors, you should be able to monitor its performance and see higher CPU resource utilization. For example, if you have an 8-core computer, you should see utilization closer to 800%.

Let's discuss haw we can make your code parallel, with the least amount of re-writing.

The `foreach` package is based on a for loop - and is designed so that one can go from serial to several forms of parallel relatively easily. There are then a number of tools one can use in the library to improve performance.

foreach - serial

The standard R `for` loop looks like this:

```
> for (i in 1:2) print(sqrt(i))
[1] 1
[1] 1.414214
```

The `foreach` operator looks similar, but returns a list of the iterations:

```
> library(foreach)
> foreach (i=1:2) %do% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214
```

foreach + doParallel

Foreach works with a variety of backends to distribute computation - `doParallel`, which allows snow- and multicore-style parallelism, and `doMPI` (not covered here).

Switching the above loop to parallel just requires registering a backend and using `%dopar%` rather than `%do%`:

```
> library(doParallel)
> registerDoParallel(3)  # use multicore-style forking
> foreach (i=1:2) %dopar% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

> stopImplicitCluster()
```

foreach + doParallel

One can also use a PSOCK cluster:

```
> cl <- makePSOCKcluster(3)
> registerDoParallel(cl)  # use the just-made PSOCK cluster
> foreach (i=1:3) %dopar% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

> stopCluster(cl)
```

Combining results

While returning a list is the default, `foreach` has a number of ways to combine the individual results:

```
> foreach (i=1:3, .combine=c) %do% sqrt(i)
[1] 1.000000 1.414214 1.732051
> foreach (i=1:3, .combine=cbind) %do% sqrt(i)
     result.1 result.2 result.3
[1,]        1 1.414214 1.732051
> foreach (i=1:3, .combine="+") %do% sqrt(i)
[1] 4.146264
> foreach (i=1:3, .multicombine=TRUE, .combine="sum") %do% sqrt(i)
[1] 4.146264
```

Most of these are self explanatory. `multicombine` is worth mentioning: by default, `foreach` will combine each new item individually. If `.multicombine=TRUE`, then you are saying that you're passing a function which will do the right thing even if foreach gives it a whole wack of new results as a list or vector - *e.g.*, a whole chunk at a time.

Summary: foreach

Foreach is a wrapper for the other parallel methods we've seen, so it inherits some of the advantages and drawbacks of each.

Use `foreach` if: - Your code already relies on `for`-style iteration; transition is easy

- You don't know if you want multicore vs. snow style `parallel` use: you can switch just by registering a different backend!
- You want to be able to incrementally improve the performance of your code.

Note that you can have portions of your analysis code use `foreach` with `parallel` and portions using the backend with apply-style parallelism; it doesn't have to be all one or the other.

Compiled code

It is possible to interface your R code with compiled code. Why would you want to do that?

- It's fast! Compiled code is always faster than interpreted code
- If you can get the slowest parts of your code into a compiled language, you can leave the rest in R
- R comes with the ability to byte-compile specific functions
- It's also possible to write your own pure C++ or Fortran code to interface with R, but it's a pain
- It's easier to use the Rcpp package, written by Dirk Eddelbuettel, Romain Francois, and others
- This package allows you to easily interface with C++ code

Byte-compiled R code

We can byte-compile specific R functions using the `compiler` package. Since R 3.4.0, loops are automatically byte-compiled before they are run, and all functions are compiled on their first or second use.

Here we're using the `enableJIT` (Just In Time compiler) function to turn off automatic byte compiling. In general, you should NOT do this. We're only doing this for the purposes of comparing speeds.

```
> library(compiler); library(microbenchmark)
> oldJIT <- enableJIT(0); n <- 1e5
> f <- function(n) { x <- 1; for (i in 1:n) x <- 1 / (1 + x) }
> lf <- cmpfun(f)
> microbenchmark(f(n), lf(n))
Unit: milliseconds
  expr       min        lq      mean    median       uq       max neval
  f(n) 13.439841 13.778255 14.309054 13.860767 14.38052 27.337693   100
 lf(n)  1.359232  1.362942  1.372936  1.364152  1.37596  1.526307   100
```

Byte-compiled R code

Some notes about the last slide:

- Byte compiling is not the same as actually compiling code, as is done with compiled languages:
    - Byte compiling creates a byte object, which is executed by a virtual machine
    - Compiled languages are compiled into machine code, which is directly used by the hardware
- Nonetheless, byte compiling can be significantly faster than running the code through the R interpreter
- If you run a function multiple times, R will automatically byte-compile it for you. Better to just byte-compile it in your utilities file.
- Automatic byte compiling can be turned off using the `enableJIT` function, though this is not recommended

Installing `Rcpp`

We're going to be doing examples with `Rcpp`. But, if you're using Windows. . .

- `Rcpp` is not a default R package; you will need to download and install it
- Because `Rcpp` compiles code (that's the point), you will need a compiler on your computer
- If you're using Linux or a Mac, you're probably OK
- On Windows, you need to go here, and download "Rtools":

https://cran.r-project.org/bin/windows/Rtools

Note that Rtools is quite large, and will require some time to download. It's probably best not to do this during class.

Using Rcpp

Once the function is defined, it will automatically be compiled, this is why it takes a moment for the `cppFunction` command to finish.

Once compiled, `Rcpp` creates an R function which links to the compiled C++ code.

```
> library(Rcpp)
> cppFunction("int times(int x, int y) {
+   int product = x * y;
+   return product;
+ }")
> times(34, 4)
[1] 136
> 34 * 4
[1] 136
```

Using `Rcpp`

Some notes about this example:

- `Rcpp` defines special C++ data types which are compatible with R data types:
  - IntegerVector, NumericVector, LogicalVector, CharacterVector
  - IntegerMatrix, NumericMatrix, LogicalMatrix, CharacterMatrix
  - Lists, DataFrames
- These data types allow the ability to deal with missing values, using the `is_na()` function

Note that you should always test your code carefully when using multiple languages. Sometimes surprises can creep in.

Making your code awesome

Some tips:

- Save your function profiling until you know that the function works correctly. Don't succumb to "premature optimization"
- Do byte compiling first. It's easy and may be good enough
- Put your byte-compiled functions in your utilities files
- Don't be afraid of `Rcpp`. Once you know how to program in one language, you're at least 80% of the way to programming in all languages
- Ask us for help, if speed becomes an issue for your productivity