Lecture outline

- Reminder: **Vectors** as basic **data types**
- **Lists** as basic **data structures**
- Relation between data tables, vectors, and lists
- Using **lists** to write **classes**
- Generic programming in R

In [1]:
```
%load_ext rmagic
```

Data table columns, R vectors, and R types

As we saw, R basic types are **vectors** and not **scalars**. They are already vectorized container! This allows for vectorized looping, branching, and function application.

The mains basic vectorized types are:

- **numeric** vectors
- **logical** vectors
- **character** vectors

and they correspond to the **columns** of a data table.

The data table columns contain the values of the population **characteristics** we are studying. Here are several equivalent names (associated with different fields) to denote these population characteristics:

- programming: **attributes**
- machine learning: **features**
- statistics: **variables**
- physics: **observables**
- mathematics: **functions**

Mathematically, a population characteristic is a function $X:\Omega\to A$ from the **population** (also called **universe**) under study to a given set of **values** A that the characteristic can take.

The values in a given data table column constitute a **sample** of the function $X:\Omega\to A$:

$$X(s_1),\, X(s_2),\, \dots, X(s_n)$$

where $s_1,\dots,s_n\in S$ and $S\subset\Omega$ is a subset of our population, called a **population sample** (which, in some case as the student grade example, can coincide with the total population).

In statistics, the **mode** of a variable corresponds roughly to the notion of **type** in programming. In statistics, one is interesteed in knowing only if the values of A are **numercial**, in which case the variable is **quantitative**, **characters**, in which case the variable is **categorical**, or **logical** (which is a special case of a categorical variable).

In programming, one needs to distinguish between integer or float, since they take different amoung of storage room in the computer memory.

R offers two functions reflecting this distinction between programming and statistics:

```
mode(x) and typeof(x),
```

the former corresponding to the **statistical mode** and the latter corresponding to the **programming type** of a given variable `x`.

In [6]:
```
%%R

X = c(1, 2, 3) ; print(mode(X))
Y = c('a', 'b') ; print(mode(Y))
Z = c(T, T, F) ; print(mode(Z))
```

```
[1] "numeric"
[1] "character"
[1] "logical"
```

Data table rows, R lists, and R classes

Lists as data table rows

Since data tables are central and statistics, and since R was designed with statistics in mind, there should be a mechanism to group R vectors, corresponding to values of certain population characteristics together into a kind of data table.

Why not use vectors to group vectors together? Each vector component would then be a vector containing the value of our statistical variable. Let's try.

In [8]:
```
%%R
midterm = c(1, 3, 4)
major = c('MATH', 'STAT', 'ECON')

table = c(midterm, major)
print(table)
```

```
[1] "1"    "3"    "4"    "MATH" "STAT" "ECON"
```

Two things happened here that are not to our taste:

1. The `c` function has **flattened** our table: what we obtained is again a vector and not a *table*!

2. The types have been also flattened out, and **converted** to the **lowest common multiple**: i.e., numbers have been interpreteed as strings, such that the resulting vector has a the **same** type for all of its components.

This reflects the facts that, as we have seen, vectors ARE types, and, as such, must contained elements of the same nature: all numbers, all character string, or all Boolean.

Luckily, R has also a **basic data structure**: the **lists**.

At contrast with vectors, **lists can have elements of different nature** for their componets (including list themselves).

> **Lists** are **heterogeneous** collections, while **vectors** are **homogeneous** collections.
>
> **Lists** represent **data table rows**, while **vectors** represent **datatable columns**!
>
> **Lists** represent **data structures**, while **vectors** represent **data types**!

Here's now how to create a list in R:

In [9]:

```
%%R

student = list(firstname='Bob', SID='1343243', Year='Sophomore', GPA=3.4,
age=23)
print(student)
```

```
$firstname
[1] "Bob"

$SID
[1] "1343243"

$Year
[1] "Sophomore"

$GPA
[1] 3.4

$age
[1] 23
```

Retrieving elements with the dollar sign notation

So lists are also **labelled** collections, as vectors, since we can assign **names** or **labels** to their elements, using the same construct as for vectors, **naming the arguments** passed to the function `list`:

```
list(name1=value1, name2=value2, etc.)
```

The function `print` prints the list components, indicating the label by suffixing it with a dollar sign, the corresponding value below, is a regular R type, that is, a vector (in our previous example with only one component.)

One can also use the **dollar notation**

```
list$element_name
```

to retrieve the corresponding element of a list.

The dollar notation is extremely close to the period notation for Python classes, allowing us to retrive the attributes of a given Python class!

In [6]:
```
%%R

print(student$firstname)
print(student$SID)
```

```
[1] "Bob"
[1] "1343243"
```

Retrieving elements using the bracket operator

R lists are very much like **Python dictionaries**, or better, like **Pandas DataFrames** with a single row.

As Python dictionaries, or Pandas DataFrame, list elements can be also accessed using the **bracket operator**:

```
list[range]
```

where `range` works exactly as for R vectors.

In [7]:
```
%%R
a = student['firstname']
print(a)
```

```
$firstname
[1] "Bob"
```

In [8]:
```
%%R
b = student[c('firstname','SID')]
print(b)
```

```
$firstname
[1] "Bob"

$SID
```

```
[1] "1343243"
```

In [9]:
```
%%R
c = student[1]
print(c)
```

```
$firstname
[1] "Bob"
```

In [10]:
```
%%R
d = student[c(1,2,3)]
print(d)
```

```
$firstname
[1] "Bob"

$SID
[1] "1343243"

$Year
[1] "Sophomore"
```

In [11]:
```
%%R
e = student[1:3]
print(e)
```

```
$firstname
[1] "Bob"

$SID
[1] "1343243"

$Year
[1] "Sophomore"
```

In [12]:
```
%%R

f = student[-2]
print(f)
```

```
$firstname
[1] "Bob"

$Year
[1] "Sophomore"

$GPA
[1] 3.4
```

```
$age
[1] 23
```

A simple data table representation using lists

Using a list, we can store the first column of our data table as a vector of a certain mode in the list first element, the second column in the list second element, and so on.

This gives us a quick and dirty way to represent a data table in R:

In [13]:
```
%%R

F = c (Bob=62, Julien=39, Julie=84)
M = c (Bob=12, Julien=34, Julie=64)
sid = c(Bob=23513, Julien=4532, Julie=5424)

grades= list(SID=sid, midterm=M, final=F)

print(grades)
```

```
$SID
   Bob Julien  Julie
 23513   4532   5424

$midterm
   Bob Julien  Julie
    12     34     64

$final
   Bob Julien  Julie
    62     39     84
```

Since, lists elements are vectors one can compute with them **in a vectorized way**, provided that the vectorized operations make sense between the list elements (for instance, it makes sense to add only numerical vectors, etc.)

To illustrate this, let us compute the total grade for each stdudent in our previous example,and add the result back to our grade table:

In [14]:
```
%%R

TG = 0.5*grades$midterm + 0.6*grades$final

grades$TG = TG
print(grades)
```

```
$SID
   Bob Julien  Julie
```

```
 23513   4532   5424

$midterm
   Bob Julien  Julie
    12     34     64

$final
   Bob Julien  Julie
    62     39     84

$TG
   Bob Julien  Julie
  43.2   40.4   82.4
```

Lists as classes

If we stop to think of it, the rows in a data table ressemble much the notion of **class instances** or **objects** that we saw in Python.

Namely, one can think as the **column labels** in a data table as the various **attributes** defining a class. In this way of thinking, the actual **data table rows** correspond to the **actual class instances**, or **objects**.

In our previous example, the list `student`, representing a data table row, encapsulates five **variables**, or better five **attributes** representing the notion (or class) *student*: Namely,

```
name, SID, Year, GPA, age
```

To summarize, we have:

- **list elements = class attributes**

R supports very much this interpretation of list elements as class attributes: Namely, R provides a function

```
attributes(x)
```

that takes a list `x` and **returns the names of the list attributes**(or elements):

In [4]:
```
%%R

student=list(name='Bob', SID='1234', Year='Sophomore', GPA=3.4, age=12)
print(student)
print(class(student))
#print(attributes(student))
```

```
$name
[1] "Bob"

$SID

[1] "1234"
```

```
$Year
[1] "Sophomore"

$GPA
[1] 3.4

$age
[1] 12

[1] "list"
```

Moreover, one accesses a list attributes very much the same way as for Python classes, except that the **period** is replaced by a **dollar sign**.

The class mechanism: attributes and constructor

Actually, R provides a few mechanisms that allow us to define **classes** using lists.

First of all there is a function

```
class(x)
```

that returns the "class" of an object:

In [5]:
```
%%R
print(class(student))
```

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

So, the **class** of our object `student` is: `list`. This is not completely satisfactory, since we'd like to define our on class: `Student`

The trick here is that the return value of the function `class(x)` is a **reference** to a special string contained in a list: the **class** string.

Defining our own class amounts to setting this **class string** to whatever we wish to:

In [9]:
```
%%R
class(student) = 'Student'
```

Now our student list is of class `student`!

In [10]:
```
%%R
print(class(student))
```

```
[1] "Student"
```

The function attributes returns now a list with two character vectors as elements:

The function attributes returns now a list with two character vectors as elements:

- the first containing the **attribute names**
- the second (of length 1) containing the **class name**

In [11]:

```
%%R
attr = attributes(student)
print(attr)
```

```
$names
[1] "name" "SID"  "Year" "GPA"  "age"

$class
[1] "Student"
```

To emulate Python classes, we are still lacking a few things. One of them is the notion of a **constructor**, that is a function that will construct objects of our class from the values we pass to it as arguement.

The way to do so in R is very simple: just write a function that does the job:

In [12]:

```
%%R

Student = function(firstname, SID, Year, GPA, age)
{
    student = list(firstname=firstname, SID=SID, Year=Year, GPA=GPA, age=a
ge)
    class(student) = 'Student'
    return(student)
}
```

Now we can construct many student objects, with always the same attribute structure thanks to our constructor:

In [13]:

```
%%R

Bob = Student('Bob', '24213', 'Freshman', 3.4, 24)
```

Let's check the attributes of our object:

In [14]:

```
%%R

print(attributes(Bob))
```

```
$names
[1] "firstname" "SID"       "Year"      "GPA"       "age"

$class
[1] "Student"
```

In [14]:
```
%%R
#print(Bob)
print(class(Bob))
```

```
[1] "Student"
```

We are still missing half of the story if we want to compete with Python classes: the **methods**.

The class mechanism: methods and generic functions

Recall that we introduced classes in Python as being a convenience offered by the language allowing us to **encapsulate**

- **data** in the form of a collection of variables: **the class attributes**
- **functions** acting naturally on this data: **the class methods**

Writing methods for our own class relies in R on **naming conventions**. This means that a method for a given classis a **regular** function, whose name follows the following convention:

```
function_name.class_name = function(object, arg1, arg1, etc.){ func
tion body }
```

Remark: The `object` argument has the same function as the `self` argument that we need to pass as first parameter to Python class methods.

For instance, let us write a `display` method for our class `Student` that will display nicely student information:

In [15]:
```
%%R

display.Student = function(student)
{
        for (attr in attributes(student)$name)
        {
            display_str = sprintf("%10s: %s\n", attr, student[[attr]])
            cat(display_str)
        }

}
```

Remark: To retrieve the **value** of the attribute `attr`, we used the **double bracket** operator

```
student[[attr]]
```

instead of the **single bracket operator**. The reason for that is the following: accessing a list element with the

- **single bracket operator** will return a list of one element containing the correponding value
- **double bracket operator** will return the value itself (i.e. here a character vector with one element)

Invoking our method now is no different than invoking any other function, since methods are just function with a special convention for their names:

In [16]:
```
%%R

display.Student(Bob)
```

```
 firstname: Bob
       SID: 24213
      Year: Freshman
       GPA: 3.4
       age: 24
```

So far, the naming convention for methods is just a good practice for book keeping. In R, methods starts to become interresant in R when used in conjunction with **generic functions**.

A **generic function**, like `print` is a function that, if applied to an object of a certain class, will lookup to find a corresponding class method named using the convention we outlined above.

For example,

```
print(student)
```

will search for a method named

```
print.Student(x)
```

defined for our class, and invoke this method if found. If not, print will invoke the method of the class `list` and print the list underlying our Student object.

To see what classes implement a method for a generic function, you can use the following command:

In [26]:
```
%%R

methods(print)
```

We see that our class does not implement this method. So when we print a Student object, in fact, the underlying list is printed:

In [21]:
```
%%R
print(student)
```

```
$firstname
```

```
[1] "Bob"

$SID
[1] "1343243"

$Year
[1] "Sophomore"

$GPA
[1] 3.4

$age
[1] 23

attr(,"class")
[1] "Student"
```

Let's implement the **method print** for the class `Student` and see how the **generic function print** is affected:

In [23]:
```
%%R

print.Student = function(student)
{
    display.Student(student)
}
```

Now, let's call again the generic `print` on a Student object:

In [22]:
```
%%R

print(student)
```

```
      name: Bob
       SID: 1234
      Year: Sophomore
       GPA: 3.4
       age: 12
```

Great! Now what if we whant to **promote our method**

```
display.Student(x)
```

to a **generic function**?

We simply need to write a function

```
display(x)
```

that will invoke the **special function**

```
UseMethod(name, x)
```

This function will

- **lookup the class** of the object `x`
- **lookup for a method** with name `name` implemented for this class
- pass the object `x` as argument to this method

In [18]:
```
%%R

display = function(object)
{
    UseMethod('display',object)
}
```

In [19]:
```
%%R
print(methods(display))
```

```
[1] display.Student
```

In [20]:
```
%%R
display(student)
```

```
    name: Bob
     SID: 1234
    Year: Sophomore
     GPA: 3.4
     age: 12
```

BREAKOUT:

Write a class `employee` with attributes

- name
- employer
- job title
- hourly rate
- number of hours worked per month

and generic functions

- print
- salary

In [32]:
```
```

In [ ]:
```
```