Bioinformática y análisis de datos ómicos

Tema 4: Análisis Bioinformáticos Complejos en R

Ignacio Varela Egocheaga
Departamento de Biología Molecular

Tema 4: Análisis bioinformáticos complejos en R

4.1 Ejecución condicional de funciones

4.2 Repeticiones o ciclos de operaciones

4.3 Generación de funciones personalizadas

4.4 Creacción de objetos personalizados

Tema 4: Análisis bioinformáticos complejos en R

4.1 Ejecución condicional de funciones

4.2 Repeticiones o ciclos de operaciones

4.3 Generación de funciones personalizadas

4.4 Creacción de objetos personalizados

Ejecución condicional de funciones

Todo lenguaje de programación incorpora una estructura que permite ejecutar una determinada función dependiendo de si se cumple o no una función

```
if (condición) {  #La condición debe evaluarse como TRUE o FALSE
      #ejecuta una o varias funciones

} else{
      #ejecuta una o varias funciones

}
```

```
x <- 3
if ( x > 2) {
      print("El número es mayor de 2")
} else {
      print("el número es menor de 2")
}
```

Anillamiento de diversas condiciones

```
x ← 1
if ( x > 2) {
      print(“El número es mayor de 2”)
} else {
      if ( x > 0) {
            print(“el número es mayor de 0”)
      } else {
            print “(el número es menor de 0”)
      }
}
```

ó

```
if ( x <2 & x > 0 ) {
      print(“El número es mayor de 0”)
} else {
      if ( x > 2) {
            print(“el número es mayor de 2”)
      } else {
            print “(el número es menor de 0”)
      }
}
```

Tema 4: Análisis bioinformáticos complejos en R

4.1 Ejecución condicional de funciones

4.2 Repeticiones o bucles de operaciones

4.3 Generación de funciones personalizadas

4.4 Creacción de objetos personalizados

bucles o *loops FOR*

Los bucles **FOR** permiten realizar una determinada función varias veces

```
for (variable in vector) {

    #ejecuta una o varias funciones
}
```

```
x <- 1:5
for ( number in x) {
    print(c("El número que estamos examinando es ",number), quote = FALSE)
}
```

Combinándolo con ejecuciones condiciones y con los comandos **next** y **break** dentro del loop permiten tener mayor control sobre la ejecución

```
for ( number in 1:7) {
  if ( number == 3)
  {
      next;
  }else{
      if ( number == 6)
      {
          break;
      } else {
          print(c("El número que estamos examinando es ",number), quote = FALSE)
      }

  }
```

bucles o *loops WHILE*

Los bucles **WHILE** permiten realizar una determinada siempre y cuando se cumpla una condición

```
while (condition) {      #TRUE o FALSE

      #ejecuta una o varias funciones
}
```

```
number <- 1
while ( number < 6) {
      print(number)
      number ← number + 1
}
```

¡¡muy importante!!

La utilización de los ciclos while simplifica mucho la sintaxis pero puede crear bucles infinitos y generar muchos problemas en la ejecución de los programas.

Tema 4: Análisis bioinformáticos complejos en R

4.1 Ejecución condicional de funciones

4.2 Repeticiones o bucles de operaciones

4.3 Generación de funciones personalizadas

4.4 Creacción de objetos personalizados

Funciones personalizadas

La estructura general de cualquier función en R es la siguiente

Nombre_ Función (parámetro 1, parámetro 2, ...)

Nosotros podemos crear nuestras propias funciones con la orden **function**

Nombre ← function(parámetro 1=DEFAULT, parámetro 2, …) {
#código
}

```
Nombre_nucleotido  <- function(i) {
      if ( i == “A” || i == “a” ){
                  print (“Adenina”)
      } else {
                  if ( i == “C” || i == “c”) {
                        print(“Citosina”)
                  } else {
                        if( i == “G” || i == “g”) {
                              print (“Guanina”)
                        } else {
                              if (  i == “T” || i == “t” ){
                                    print (“Timina”)
                              } else {
                                    print (“Nucleótido no reconocido”)
                              }
                        }
                  }
      }
```

Funciones personalizadas

Las funciones generalmente devuelven un objeto que podemos asignar a una variable.

```
Dividir_dos_numeros  <- function(a, b) {
            division <- a/b
            return(division)
      }

c ← Dividir_dos_numeros(4,2)  # c → 2
```

Podemos asignar parámetros opcionales con un valor por defecto .

```
Dividir  <- function(a, b = 4) {
            division <- a/b
            return(division)
      }
c ←Dividir(4) # c → 1
d ← Dividir(4,3) # d  → 1.33
```

sapply para sustituir bluces FOR

El poder definir funciones personalizadas nos permite sustituir los bucles for por la función sapply en vectores.

```
x <- c(1:5)
for ( number in x) {
      print(c("El número que estamos examinando es ",number), quote = FALSE)
}
```

ó

```
imprimir ← function (a) {
      print(c("El número que estamos examinando es ",number), quote = FALSE)
}
sapply(c(1:5),imprimir)
```

Funciones personalizadas

Ejemplo de función para realizar reversa complementaria de una secuencia de DNA.

```
rev_comp ← function(a) {

        seq_vector ← strsplt(a,"")[[1]] #convertir frase en vector
        rev_vector ← rev(seq_vector) #revertimos el vector

                comp <- function(a){  # Función para generar la base complementaria
                        if ( a == "A" | a == "a")
                        {
                                return("T")
                        } else {
                          if ( a == "C" | a == "c")
                          {
                                return("G")
                          } else{
                            if ( a == "G" | a == "g")
                         {
                                return("C")
                         } else {
                           if ( a == "T" | a == "t")
                           {
                                        return("A")
                            } else {
                                        return(a)
                           }
                        }
                      }
                    }
                }

        revcomp_vector ← sapply(rev_vector, comp)  # generar complementaria en todas las bases del vector
        return(paste(revcomp_vector,collapse="") #generar de nuevo una frase a partir del vector

}
```

Tema 4: Análisis bioinformáticos complejos en R

4.1 Ejecución condicional de funciones

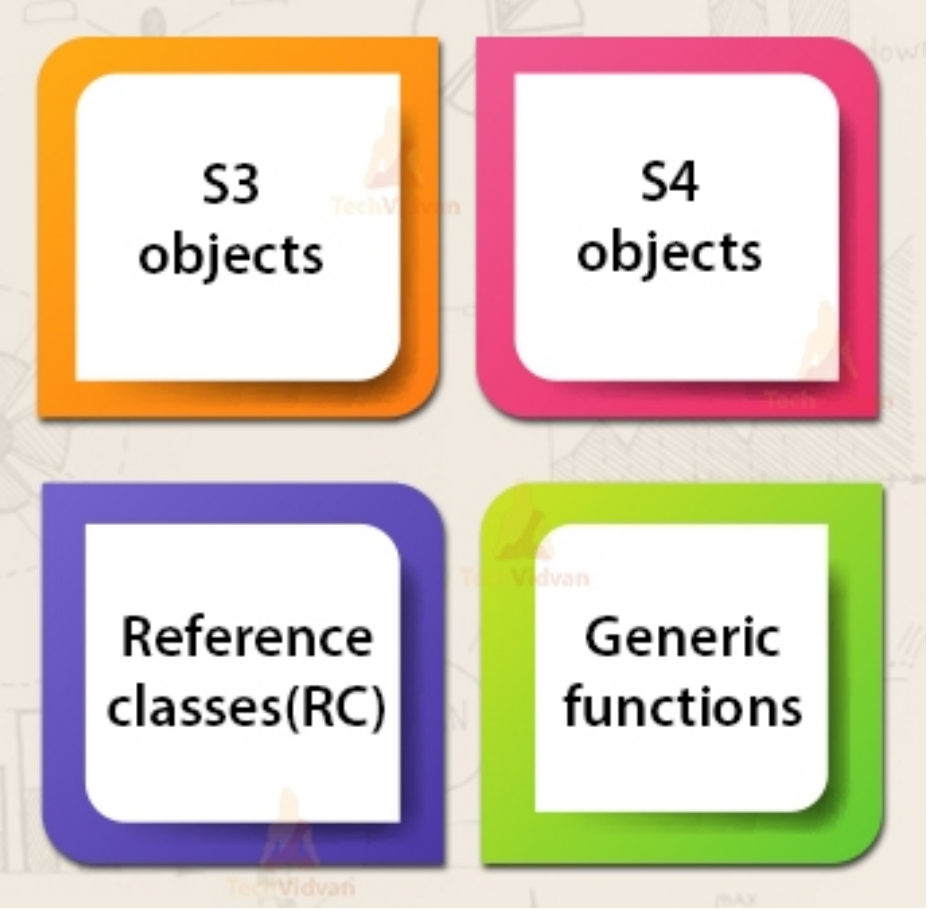
4.2 Repeticiones o bucles de operaciones

4.3 Generación de funciones personalizadas

4.4 Creacción de objetos personalizados

¿Qué es un objeto?

Un objeto se define como un conjunto de información estructurada de una manera concreta sobre el que se pueden ejecutar una serie de funciones específicas.



https://techvidvan.com/tutorials/r-object-oriented-programming/

Crear un objeto personalizado

La **CLASE** es un grupo de objetos del mismo tipo que se comportan igual ante las mismas funciones.

Un objeto **S3** es básicamente un tipo especial de lista que se asigna a una clase concreta personalizada. La asignación se realiza con la función **class**

```
Transcrito1 ← list(name="TP53-001",gene="TP53",sequence="ACTTCGC...")
class(Transcrito1) ← "Transcrito"
```

La distinta información del transcrito se puede extraer y/o modificar utilizando, como en el caso de las listas, el caracter $

```
Transcrito1$gene # TP53
Transcrito1$name ← "TP53-002" #modificamos la información en $name
```

Definir la actuación de una función genérica sobre nuestro objeto

Las funciones que hemos utilizado hasta ahora son funciones genéricas que se comportan de manera distinta dependiendo del objeto sobre el que actúan.

```
print ← function(object) {

      UseMethod("print")
}
```

Cada objeto tiene una función print específica que está definida para cada objeto y que se llama cada vez que se ejecuta la función print.

```
methods(print)
```

```
[49] print.check_S3_methods_needing_delayed_registration* print.check_so_symbols*
[51] print.check_T_and_F*                                 print.check_url_db*
[53] print.check_vignette_index*                          print.checkDocFiles*
[55] print.checkDocStyle*                                 print.checkFF*
[57] print.checkRd*                                       print.checkRdContents*
[59] print.checkReplaceFuns*                              print.checkS3methods*
[61] print.checkTnF*                                      print.checkVignettes*
[63] print.citation*                                      print.codoc*
[65] print.codocClasses*                                  print.codocData*
[67] print.colorConverter*                                print.compactPDF*
[69] print.condition                                      print.connection
[71] print.CRAN_package_reverse_dependencies_and_views*   print.data.frame
[73] print.Date                                           print.default
[75] print.dendrogram*                                    print.density*
[77] print.difftime                                       print.dist*
[79] print.Dlist                                          print.DLLInfo
[81] print.DLLInfoList                                    print.DLLRegisteredRoutines
```

Definir la actuación de una función genérica sobre nuestro objeto

Podemos entonces definir el comportamiento de la función print en nuestro objeto

```
print.Transcrito ← function(obj){
      cat ("Objeto tipo S3 representando el transcrito ", obj$name, " del gen ", obj$gene)
}
```

Ahora cada vez que llamemos la función print o simplemente escribamos el nombre del objeto (otra manera de llamar print), esta función se comportará de manera distinta a como lo hacía con una lista normal.

```
print(Transcrito1)
Transcrito1
```

Objeto tipo S3 representando el transcrito TP53-001 del gen TP53

Crear un objeto personalizado

Los objetos tipo **S4** son un poco más complicados que los S3 y se parecen más a los objetos creados en otros tipos de lenguajes como C++ o JAVA. En los objetos S4 primero hay que definir que información y que tipo de información se va a guardar en el objeto. Requiere, por lo tanto, algo más de planificación.

La función **setClass()** define el objeto tipo S4 que queremos crear con la información y el tipo de información que va a contener.

setClass(“Gen”, slots=list(cromosoma=”character”, inicio=”numeric”,final=”numeric”,nombre =”character”))

La función **new()** que en otros lenguajes se denomina constructor, permite crear nuevos objetos de esta clase

Gen1 ← **new**(“Gen”,nombre=”TP53”,cromosoma=”Chr17”, inicio=7661779, final=7687538)

Objetos S4 avanzado

Los distintos elementos que contiene el objeto S4 utilizando el caracter @

```
Gen1@nombre # TP53
Gen11@cromosoma ← "Chr16" #modificamos la información en @cromosoma
```

A diferencia de los objetos S3, no podemos ampliar directamente el número de slots que contiene el objeto ya definido. En el caso de objetos S4 debemos crear otra clase "hija"o **subclase** de la primera con otro nombre que contiene todo lo de la clase "madre" más las características que le queramos añadir

```
setClass("Gen_plus",
        contains= "Gen",
        prototype=(frame=0),
        slots=list(frame="numeric"),
)
```

La función **contains()** permite heredar las características de la clase "Gen".
La función **prototype()** permite asignar valores por defecto en caso de que no se suministren al crear el objeto

Definir funciones genéricas en objetos S4

De manera similar a lo que pasa en los objetos tipo S3, podemos definir el comportamiento de las funciones genéricas así como crear funciones nuevas sobre nuestro nuevo objeto. Para ello usamos la función **setMethod()**

```
SetMethod ( “show”,”Gen”,
      function(obj) {
                  cat( “Objeto que refleja el gen ”, obj@nombre)
            }
)
```

```
show(Gen1)
```

Objeto que refleja el gen TP53

Objetos dentro de objetos

Entre los componentes de un objeto S4 podemos introducir otros objetos S4 a modo de muñecas rusas. Esto nos permite hacer objetos cada vez más complejos para guardar la información de una manera muy organizada

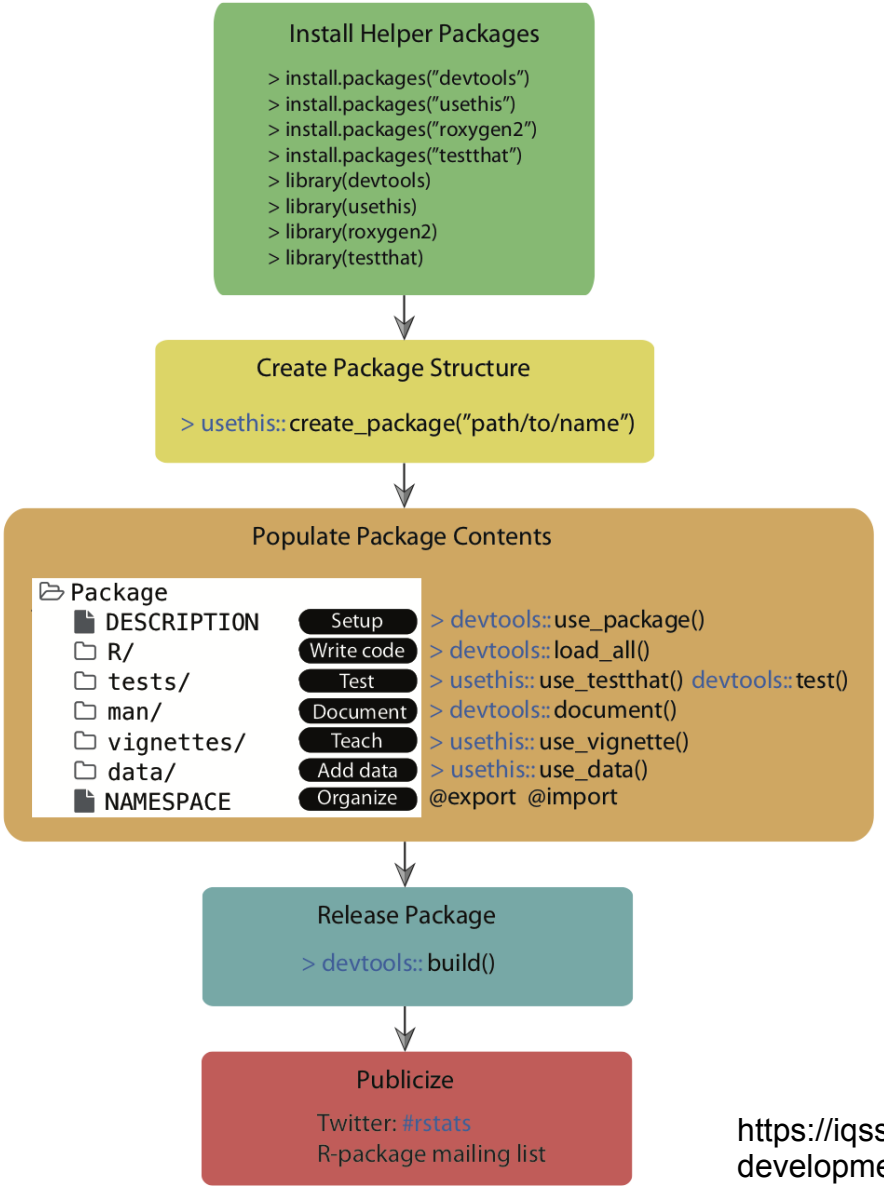
setClass(“Transcrito”, slots=list(gene=”character”, secuencia=”charcater”,nombre =”character”))
Transcrito1 ← new(“Transcrito”, gene=”TP53”, nombre=”TP53-001”, secuencia=”ACTTGCCG…”)

setClass(“Gen”, slots=list(chromosoma=”character”, nombre =”character”,transcrito=”Transcrito”))
Gene1 ← new(“Gen”, chromosoma=”Chr17”, nombre=”TP53”, transcrito=Transcrito1)

Gene1@transcrito # objeto S4 Transcrito1

Generando tus propios paquetes

Se puede guardar todo el código de nuevos objetos y funciones en una estructura de directorios concretas y crear con eso un paquete de código que puedes reutilizar siempre que quieras. Existen funciones especiales para simplificar el proceso.



https://iqss.github.io/dss-rbuild/package-development.html

Sintaxis en diferentes lenguajes de programación

La mayoría de los lenguajes de programación tienen estructuras de datos y bucles que funcionan de manera similar. Aunque en cada lenguaje es necesario invertir cierto tiempo en entender sus peculiaridades, una vez dominado un lenguaje, es fácil aprender nuevos lenguajes de programación.

Perl

```perl
my @numbers = ( 1, 3, 7 );
print "\$number before: $number\n";
foreach $number ( @numbers ) {
    print "\$number is $number\n";
    unshift @numbers, "Added later";
    }
```

Python

```python
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

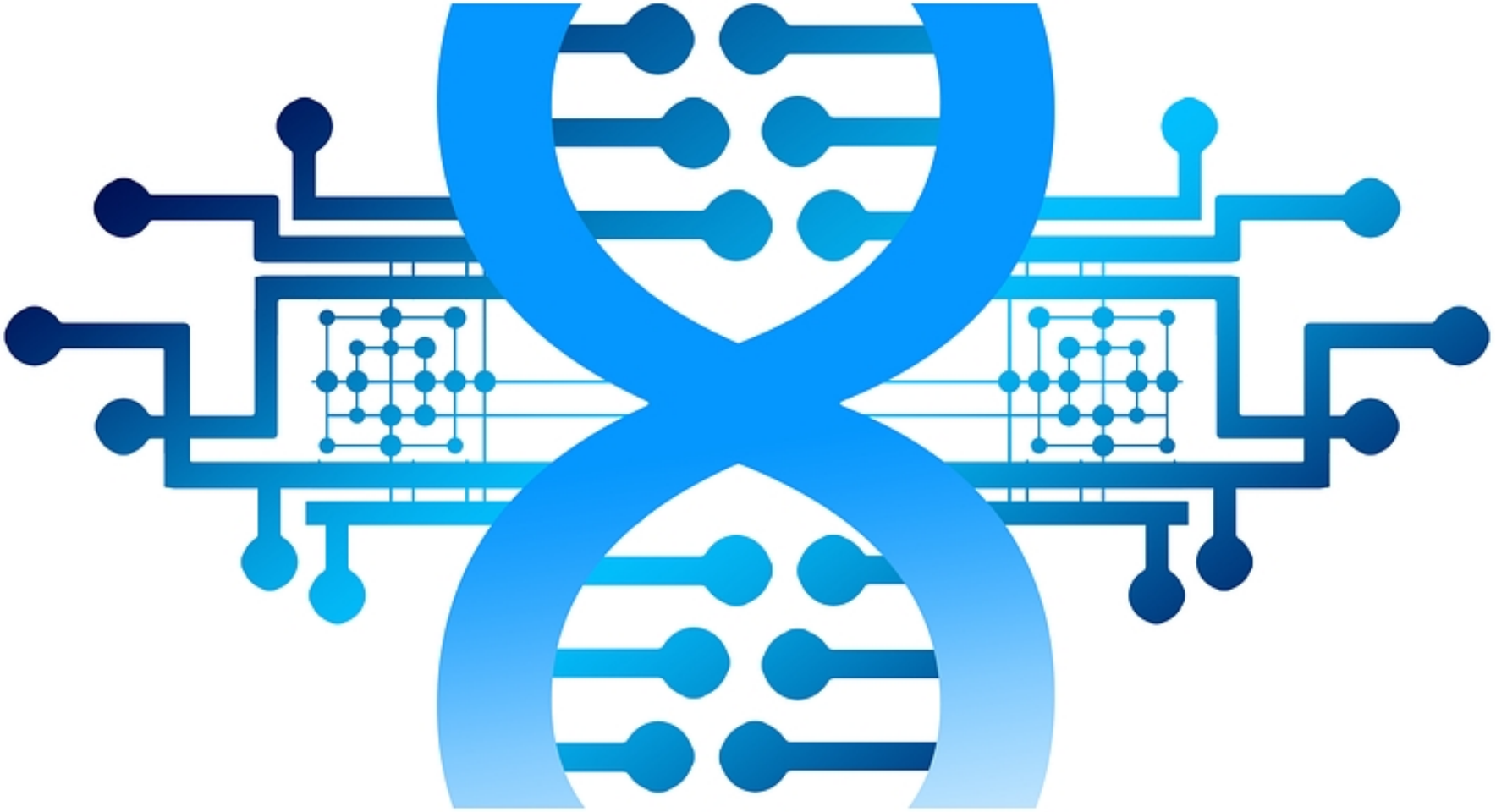
Java

```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

C++

```cpp
int array[]={1,4,7,4,8,4};
cout<<"The elements are: ";
for(auto var : array)
{
    cout<<var<<" ";
}
```

Bioinformática y análisis de datos ómicos



Tema 4: Análisis bioinformáticos complejos en R

ignacio.varela@unican.es