

S3 and S4 Objects

Department of Biostatistics and Bioinformatics

Steve Pittard wsp@emory.edu

April 27, 2016

Motivations

The plot function is very “talented”. Explain the following behavior.

```
# This gives us a scatterplot
```

```
plot(mtcars$wt,mtcars$mpg)
```

```
# We create a table and plot somehow knows how to deal with. How ?
```

```
mytable <- table(mtcars$am,mtcars$cyl)
plot(mytable)
```

```
# We create a Linear Modeling object and plot knows how to
# plot it. How ?
```

```
mylm <- lm(mpg~wt, data=mtcars)
plot(mylm)
```

Motivations

And the mystery continues.....

```
# This gives us a pairs plot
```

```
plot(mtcars)
```

```
# This will plot a histogram
```

```
myhisto <- hist(mtcars$mpg)  
plot(myhisto)
```

```
# We create a principal components object
```

```
myprcomp <- prcomp(~ Murder + Assault + Rape,  
                    data = USArrests, scale = TRUE)  
plot(myprcomp)
```

Objects

How does it do all of this ? First. Let's go back to some basics. Everything in R is an object. Functions are objects and functions can return objects. These objects belong to a class.

```
mylm <- lm(mpg~wt,data=mtcars)
class(mylm)
[1] "lm"
```

```
#
```

```
mytable <- table(mtcars$am, mtcars$cyl)
class(mytable)
[1] "table"
```

```
#
```

```
class(mtcars)
[1] "data.frame"
```

S3 Objects

It turns out that the `plot` function is not just one command. It's an **S3 class**. Usually if we type a function's name at the prompt we can see the source code but in this case we don't get that.

```
plot
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x7f861b133228>
<environment: namespace:graphics>
```

```
> methods(plot)
[1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
[4] plot.default        plot.dendrogram*    plot.density*
[7] plot.ecdf           plot.factor*        plot.formula*
[10] plot.function       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.infant          plot.isoreg*
[16] plot.lm*            plot.medpolish*     plot.mlm*
[19] plot.ppr*           plot.prcomp*        plot.princomp*
[22] plot.profile.nls*   plot.spec*          plot.stepfun
[25] plot.stl*           plot.table*         plot.ts
[28] plot.tskernel*      plot.TukeyHSD*
```

Non-visible functions are asterisked

S3 Objects and Methods

Plot is actually an S3 object that has many “methods” at it’s disposal. Notice that there is a plot method for many scenarios.

```
plot
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x7f861b133228>
<environment: namespace:graphics>

> methods(plot)
[1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
[4] plot.default        plot.dendrogram*    plot.density*
[7] plot.ecdf           plot.factor*        plot.formula*
[10] plot.function       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*    plot.infant          plot.isoreg*
[16] plot.lm*            plot.medpolish*     plot.mlm*
[19] plot.ppr*           plot.prcomp*        plot.princomp*
[22] plot.profile.nls*   plot.spec*          plot.stepfun
[25] plot.stl*           plot.table*         plot.ts
[28] plot.tskernel*      plot.TukeyHSD*
```

Non-visible functions are asterisked

S3 Objects and Methods

Note that the **summary** function is similar. It can handle a large number of objects.

```
summary
function (object, ...)
UseMethod("summary")
<bytecode: 0x7f861bf83708>
<environment: namespace:base>
```

```
> methods(summary)
[1] summary.aov                summary.aovlist*          summary.aspell*
[4] summary.connection         summary.data.frame        summary.Date
[7] summary.default            summary.ecdf*             summary.factor
[10] summary.glm                summary.infl*             summary.lm
[13] summary.loess*             summary.manova             summary.matrix
[16] summary.mlm*               summary.nls*              summary.packageStatus*
[19] summary.PDF_Dictionary*    summary.PDF_Stream*       summary.POSIXct
[22] summary.POSIXlt            summary.ppr*              summary.prcomp*
[25] summary.princomp*          summary.proc_time         summary.srcfile
[28] summary.srcref             summary.stepfun           summary.stl*
[31] summary.table              summary.tukeysmooth*
```

Non-visible functions are asterisked

S3 Objects and Methods

So if we use the **summary** function on the `mtcars` dataframe we are actually using the **summary.data.frame** function. We could call the **summary.data.frame** function directly if we wanted to.

```
grep("data.frame", methods(summary), value=TRUE)  
[1] "summary.data.frame"
```

```
summary(mtcars)
```

```
# Same as
```

```
summary.data.frame(mtcars)
```

```
# Check to see if they are identical
```

```
identical(summary(mtcars), summary.data.frame(mtcars))  
[1] TRUE
```


S3 Objects and Methods

So if we have an object of class “lm” does plot know how to deal with it ?
We already know it can.

```
mylm <- lm(mpg~wt, data=mtcars)
class(mylm)
[1] "lm"
```

```
methods(plot)
[1] plot.acf*          plot.data.frame*   plot.decomposed.ts*
[4] plot.default       plot.dendrogram*   plot.density*
[7] plot.ecdf          plot.factor*        plot.formula*
[10] plot.function      plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.infant          plot.isoreg*
[16] plot.lm*           plot.medpolish*     plot.nlm*
[19] plot.ppr*          plot.prcomp*        plot.princomp*
[22] plot.profile.nls*   plot.spec*          plot.stepfun
[25] plot.stl*          plot.table*         plot.ts
[28] plot.tskernel*     plot.TukeyHSD*
```

```
grep("lm",methods(plot),value=T)
[1] "plot.lm"          "plot.nlm"         "plot.ridgelm"
```

S3 Objects and Methods

- When we give the **plot()** function an object of a certain type, the **generic** plot function will then try to find a plot method that knows how to deal with that object type.
- It does this for you. In this case it finds the command **plot.lm()** method (which is a function) to do the work.
- You don't know what's going on and in most cases don't need to unless of course you are writing your own objects.

```
mylm <- lm(mpg~wt, data=mtcars)
class(mylm)
[1] "lm"
```

```
plot(mylm)      # Plot finds the right method based on the class of the object
```

Object Oriented Programming - OOP

OOP is a very popular and useful idea in computer science although the implementation of OOP can be different across various languages. Some things to keep in mind:

- Objects represent structured data (e.g. data frames, vectors, lists)
- We build objects using class definitions (implicit and/or explicit)
- Classes can have well defined attributes or in R 'slots'
- Classes can usually inherit from other classes
- R has three OOP implementations: S3, S4, and Reference Classes
- The base installation of R has many S3 classes
- Some say S4 is the best OOP to use (BioConductor requires it)
- Yet many developers continue to develop using S3

Object Oriented Programming - OOP

- Everything in R is a self describing object

```
class(23)
[1] "numeric"
```

```
x <- matrix(1:16,4,4)
class(x)
[1] "matrix"
```

```
class(mtcars)
[1] "data.frame"
```

- You can say that every object in R belongs to a class
- Sometimes we use objects and classes as synonyms but technically that's not true
- Objects represent an **instance** of a class
- Functions that operate on Class Objects are called **methods**

Object Oriented Programming - OOP

- Classes allow us to combine data and structure into a single idea
- Objects are realizations of a Class definition
- Consider **Homo Sapiens** as a class definition
- You and I are specific instances of that class

```
John <- HomoSapien(name="John",age=33,height=72,weight=170,race="caucasian")
```

```
Mary <- HomeSapien(name="Mary",age=22,height=62,weight=132,race="american_indi
```

- Note that name, age, height, weight, and race are **properties** or **attributes** of the Homo Sapien class.
- There are others of course but we restrict our attention to just a few of them
- The **race** attribute won't ever change whereas weight, height, and age will

Object Oriented Programming - OOP

- Every class has a **Constructor** - a way to make an instance of a class
- In R we use functions to help us create instances of a class
- Using a constructor function helps create a valid object

```
aHomoSapien <- function(name,age,height,weight,race) {  
  human <- list(name=name,age=age,height=height,weight=weight,race=race)  
  class(human) <- "homosapien"  
  return(human)  
}  
  
john <- aHomoSapien(name="John",age=33,height=72,weight=190,race="caucasian")  
class(john)  
[1] "homosapien"
```

- So **john** is an object of class **homosapien**
- So what do we do with john ? Well not much at this point

Object Oriented Programming - OOP

How do we **access** values in the object. It's easy. We can use the **\$** symbol:

```
john$name  
[1] "John"
```

```
john$age  
[1] 33
```

```
john$race  
[1] "caucasian"
```

```
john[1:2]    # John is actually a list so we can do this  
$name  
[1] "John"
```

```
$age  
[1] 33
```

However it is generally recommended to use an **Accessor** function to retrieve or set object values. This makes it easier for the user and easier for the developer to make changes to the underlying object structure.

Object Oriented Programming - OOP

Consider a more interesting class where we track John's age and weight over time:

```
age <- c(33,35,37,39,41,43,45)
weight <- c(172,178,181,185,192,200,205)

aHomoSapien <- function(name,age,height,weight,race) {
  human <- list(name=name,info=data.frame(age=age,weight=weight),
               height=height,race=race)
  class(human) <- "homosapien"
  return(human)
}

john <- aHomoSapien(name="John",age=age,height=72,
                   weight=weight,race="caucasian")

class(john)
[1] "homosapien"
```


Object Oriented Programming - OOP

- Let's do the same for Mary

```
age <- c(33,35,37,39,41,43,45)
```

```
weight <- c(118,121,132,119,111,128,132)
```

```
mary <- aHomoSapien(name="Mary",age=age,height=62,  
                    weight=weight,race="caucasian")
```

```
class(mary)  
[1] "homosapien"
```

Object Oriented Programming - OOP

- Now we could develop some interesting **methods** for analyzing this instance or any like it.
- Some common generic methods involve plotting, summarizing, or printing.
- In R the **plot**, **summary**, and **print** commands are **generic** functions that behave differently based on the class of the argument(s) passed

```
> plot
function (x, y, ...)
  UseMethod("plot")
<bytecode: 0x7f9251b80828>
<environment: namespace:graphics>
```

```
> summary
function (object, ...)
  UseMethod("summary")
<bytecode: 0x7f9252e08468>
<environment: namespace:base>
```

Object Oriented Programming - OOP

```
> methods(plot)
[1] plot.acf*           plot.data.frame*
[3] plot.decomposed.ts* plot.default
[5] plot.dendrogram*    plot.density*
[7] plot.ecdf            plot.factor*
[9] plot.formula*        plot.function
[11] plot.hclust*         plot.histogram*
[13] plot.HoltWinters*    plot.isoreg*
[15] plot.lm*             plot.medpolish*
[17] plot.mlm*            plot.newnorm
[19] plot.ppr*            plot.prcomp*
[21] plot.princomp*       plot.profile.nls*
[23] plot.shingle*        plot.spec*
[25] plot.stepfun         plot.stl*
[27] plot.table*          plot.tree
[29] plot.trellis*        plot.ts
[31] plot.tskernel*       plot.TukeyHSD*
```

Non-visible functions are asterisked

Object Oriented Programming - OOP

Why is this useful ?

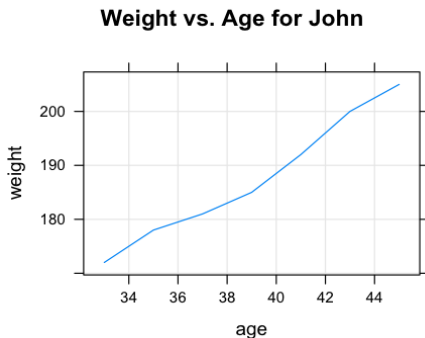
- Well R users are familiar with the `plot` function so for the non-programmer they can count on the `plot` function working with objects of class `homosapien`
- We can alter the underlying **`plot.homosapien`** without our end users knowing about it.
- We can extend any of the common generic functions including **`summary`** and **`print`**

Object Oriented Programming - OOP

Let's write a plot function specific to objects of class **homosapien**

```
plot.homosapien <- function(obj) {  
  library(lattice)  
  hold <- obj$info  
  title <- paste("Weight vs. Age for",obj$name,sep=" ")  
  xyplot(weight~age,data=hold,main=title,type=c("l","g"))  
}
```

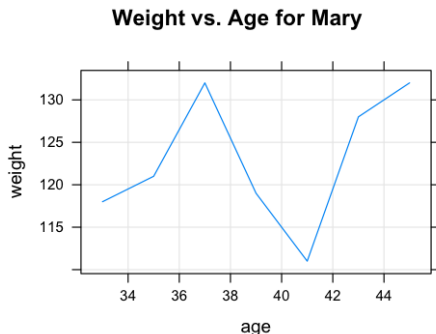
```
plot(john)
```



Object Oriented Programming - OOP

This function will work for Mary also since she too is of class `homosapien`. Note that we do NOT have to explicitly call **`plot.homosapien`** since **`plot`** is a generic function that will **dispatch** the work to the plot function defined for handling objects of type **`homosapien`**

```
plot(mary)
```



Object Oriented Programming - OOP

Let's define a **summary** method. This is common for most R objects.

```
summary.homosapien <- function(obj) {  
  str <- paste(obj$name,"had a mean weight of",  
              round(mean(obj$info$weight),2),"lbs over a period of",  
              max(obj$info$age)-min(obj$info$age),"years",sep=" ")  
  print(str)  
}
```

```
summary(john)  
[1] "John had a mean weight of 187.57 lbs over a period of 12 years"
```

```
summary(mary)  
[1] "Mary had a mean weight of 123 lbs over a period of 12 years"
```

S3 Objects and Methods - generic functions

- We can create a generic function that can then handle a variety of objects based on a given object's type. (We still have to create the functions for a specific type)
- Like with the **plot()** and **summary()** functions.
- Think of plot and summary as a generic function that handles specific objects based on their class by **dispatching** the work to more specific functions

```
mylm <- lm(mpg~wt, data=mtcars)
class(mylm)
[1] "lm"
```

```
plot(mylm)    # Plot finds the right method based on the class of the object
```


Object Oriented Programming - OOP

We can define our own generic function. For example let's come up with a function that extracts a specified range of age and weight information from the object.

```
getRange <- function(obj,start,end,...) {  
  UseMethod("getRange")  
}
```

```
getRange.homosapien <- function(obj,start,end,...) {  
  holdf <- obj$info  
  retdf <- holdf[holdf$age >= start & holdf$age <= end,]  
  return(retdf)  
}
```

```
getRange(john,35,41)  
  age weight  
2  35    178  
3  37    181  
4  39    185  
5  41    192
```

S3 Classes

So I've gotten pretty far into this without any formal definitions. There are three types of classes in R: S3, S4, and Reference Classes.

- S3 has been around for quite a while and there are many S3 classes that come with R. It's said to be “quick and dirty”
- To create an S3 class we setup an object and set it's class attribute to reflect the new class name
- We then define a generic function (if it hasn't already been done) of the form **generic.class**.
- If it's **plot**, **print**, or **summary** then the generic has already been defined for us. We just write the **generic.class** function

S3 Classes

- We can conveniently extend the capabilities of an existing function that users know (e.g. `plot` or `getHist`) without the user even knowing about it.
- That generic function can grow over time to include related types of data much in the same way the `plot` and summary functions grew over time to handle objects of different types (e.g. `tables`, `lm`, `glm`, `prcomp`, etc)
- If desired, users themselves can locally extend a generic function to accommodate new objects of their own making (though generally they don't). I did just to show you that it could be done.
- If you anticipate turning over your functions to a user unfamiliar with R programming they will greatly appreciate the existence of your generic function so they don't have to remember which function to call for what class of data.

S3 Classes

- S3 Classes are easy to create and use but they lack error checking. We can create a new S3 class and populate it with junk and get away with it.

```
age <- c(33,35,37,39,41,43,45)
weight <- c(172,178,181,185,192,200,205)

aHomoSapien <- function(name,age,height,weight,race) {
  human <- list(name=name,info=data.frame(age=age,weight=weight),
               height=height,race=race)
  class(human) <- "homosapien"
  return(human)
}

john <- aHomoSapien(name="John",age=age,height="XX",
                   weight=weight,race=23)
```

- No errors were generated despite the fact that **height** should be numeric and **race** should be a character string

S3 Classes

Function Name	Purpose
<code>str(object)</code>	Display internal object structure
<code>is(obj)</code>	Test relationship between object and class
<code>is(obj,"class")</code>	Is obj in "class" ?
<code>class(x)</code>	Show classes that x belongs to
<code>methods(class="class")</code>	Show all methods for "class"
<code>unclass(x)</code>	Remove all classes for x

Table: Important S3 Functions

An S3 Crime Class

- Okay let's read in the Chicago crime data from the vizualization lecture and use it to build an object of type "crime".

```
chi <- read.csv("chi_crimes.csv",header=T,sep="," ,stringsAsFactors=FALSE)
chi <- chi[complete.cases(chi),] # Get rid of incomplete cases
library(lubridate)
chi$Date <- parse_date_time(chi$Date,'%m/%d/%Y %I:%M:%S %p')
```

- So now we make an object/instance of "crime"

```
makecrime <- function(x,y,z) {
  crimes <- list(id=x, source=y, data=z)
  class(crimes) <- "crime"
  return(crimes)
}
```

```
chicrimes <- makecrime("Chicago,IL","https://data.cityofchicago.org/",chi)
```

An S3 Crime Class

```
print.crime <- function(object) {
  header <- paste("CITY,STATE:",object$id,"SOURCE:",object$source,sep=" ")
  cat(header,"\n\n")
  str(object$data,0)
}

plot.crime <- function(object,...) {
  stopifnot(require(dplyr))
  stopifnot(suppressPackageStartupMessages(require(googleVis)))
  df <- object$data
  df$Date <- as.Date(df$Date,format="%m/%d/%Y") # We need only the day of year
  df %>% group_by(Date) %>% summarize(count=n()) -> dfout

  Cal <- gvisCalendar(dfout, datevar="Date", numvar="count",
    options=list(width=900,height=600,
      title="Daily Crime report", height=320,
      calendar="{yearLabel: { fontName: 'Times-Roman',
        fontSize: 32, color: '#1A8763', bold: true},
        cellSize: 13,
        cellColor: { stroke: 'red', strokeOpacity: 0.2 },
        focusedCellColor: {stroke:'red'}}")
  plot(Cal)
}
```

An S3 Crime Class

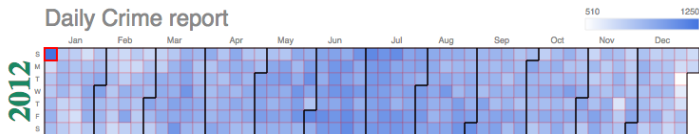
```
print(chicrimes)
```

```
CITY,STATE: Chicago,IL SOURCE: https://data.cityofchicago.org/
```

```
'data.frame': 331980 obs. of 22 variables:
```

```
# Now plot
```

```
plot(chicrimes)
```



S4 Classes

- S4 classes attempts to provide more rigor.
- We define our classes with an associated representation that must be adhered to so we can validate new instances of that class.
- Of course this means we have to go to more effort when creating classes and associated methods.
- We create a new S4 class by using the **setClass** function:

```
setClass("homosapien", representation(name="character",  
                                       info="data.frame",  
                                       height="numeric",  
                                       race="character") )
```

```
getSlots("homosapien")
```

name	info	height	race
"character"	"data.frame"	"numeric"	"character"

S4 Classes

We create a new S4 class by using the **setClass** function:

```
age <- c(33,35,37,39,41,43,45)
```

```
weight <- c(172,178,181,185,192,200,205)
```

```
john <- new("homosapien",name="John",  
           info=data.frame(age=age,weight=weight),  
           race="caucasian")
```

```
slotNames(john)  
[1] "name"    "info"    "height"  "race"
```

```
john@name  
[1] "John"
```

```
john@info  
  age weight  
1  33    172  
2  35    178  
3  37    181  
4  39    185  
5  41    192  
6  43    200  
7  45    205
```

S4 Classes

The S4 equivalent to S3's **print** method is the **show** method

```
setMethod(f = "show", signature = "homosapien",  
          definition = function(object) {  
            cat("Name: ",object@name,"\n")  
            print(object@info)  
          })
```

```
[1] "show"
```

```
> john
```

```
Name:  John
```

	age	weight
1	33	172
2	35	178
3	37	181
4	39	185
5	41	192
6	43	200
7	45	205

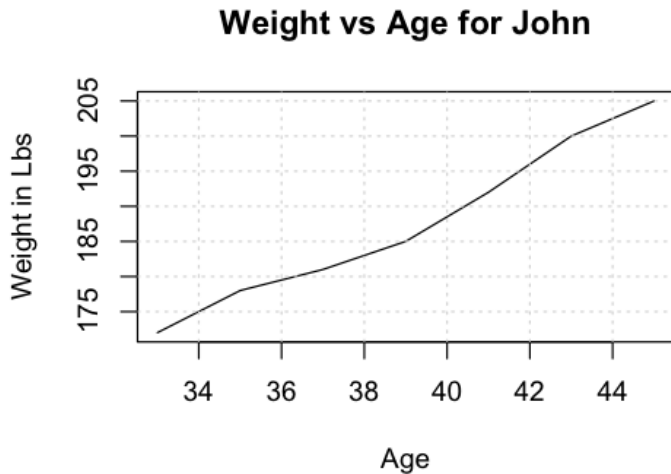
S4 Classes

Let's define an S4 plot method to handle the plotting of a Homo Sapien object. As with S3 if we are extending the capability of an existing generic function which is in this case the **plot** function. In S4 we use the **setMethod** function to add in our own definition of **plot**

```
setMethod(f = "plot", signature = "homosapien",
  definition = function(x,y,...) {
    main <- "Weight vs Age"
    xlab <- "Age"
    ylab <- "Weight in Lbs"
    name <- x$name
    hold <- x$info
    x <- hold$age
    y <- hold$weight
    main <- paste(main,"for",name,sep=" ")
    plot(x,y,main=main,type="l",xlab=xlab,ylab=ylab)
    grid()
  })
```

S4 Classes

```
plot(john)
```



S4 Classes

- So let's define our own generic function. We'll create an **Accessor** function called **getRange()** just as we did with the S3 Homo Sapien object above.
- It's considered a best practice to provide users with an accessor function because they then do not have to memorize the underlying details of the object - we might decide to change the slotnames for example.
- As the developers of the class we can change the underlying class implementation without disturbing the users.
- As long as users are using the accessor function (and we keep it updated to match any changes we make to the object) then all will be well.

S4 Classes

```
setGeneric("getRange",function(object,start,end) standardGeneric("getRange") )

setMethod("getRange","homosapien",
  function(object,start,end) {
    holdf <- object@info
    retdf <- holdf[holdf$age >= start & holdf$age <= end,]
    return(retdf)
  })

getRange(john,35,41)
  age weight
2  35    178
3  37    181
4  39    185
5  41    192
```

S4 Classes

That was okay but why not return an object of type **homosapien** so we can take advantage of the **plot** function we created ?

```
setGeneric("getRange",function(object,start,end) standardGeneric("getRange") )

setMethod("getRange","homosapien",
  function(object,start,end) {
    holdf <- object@info
    retidf <- holdf[holdf$age >= start & holdf$age <= end,]
    newhs <- new("homosapien",name="John",
                 info=retidf,race="caucasian")
    return(newhs)
  })

plot(getRange(john,35,41))
```


S4 Classes

With S3 classes we could give data of most any type and it would not complain. This is a weakness of S3. S4 objects use what is in the **representation** argument to check input types.

```
setClass("homosapien", representation(name="character",
                                       info="data.frame",
                                       height="numeric",
                                       race="character") )
```

```
age <- c(33,35,37,39,41,43,45)
weight <- c(172,178,181,185,192,200,205)
```

```
john <- new("homosapien",name="John",
            info=data.frame(age=age,weight=weight),
            race="caucasian")
```

That was okay but check this out:

```
john <- new("homosapien", name=23,
            info=data.frame(age=age,weight=weight),race="caucasian")
```

```
Error in validObject(.Object) :
invalid class ``homosapien'' object: 1: invalid object for slot ``name'' in class
``homosapien'': got class ``numeric'', should be or extend class ``character''
```

S4 Classes

We saw earlier that if we try to create an instance of an S4 class and we don't match what is in the **representation** field then it throws out an error. Actually we should create our own validation function.

```
setClass("homosapien", representation(name="character",
                                      info="data.frame",
                                      height="numeric",
                                      race="character") )
```

```
setValidity("homosapien",
  function(object) {
    retval <- TRUE
    if (!is.character(object@name)) {
      print("Name should be character")
      retval <- FALSE
    }
    if ( length(names(object@info)) != 2 ) {
      print("Data frame is not valid")
      retval <- FALSE
    }
    return(retval)
  }
)
```

```
plot(tree1)
```