S3 and S4 Classes

Wickham: http://adv-r.had.co.nz/OO-essentials.html

Object oriented programming is based on the idea that data can be encapsulated in a structure that is known to the system to have certain properties. This structure is called a `class`. Classes can have a hierarchical nature in that they can be formed from inheriting properties from other classes. Because classes have known properties, functions with generic sounding names can be written to have different behavior depending on the class. These functions are refered to as `methods`. Common examples in R are `print()`, `plot()`, `summary()`.

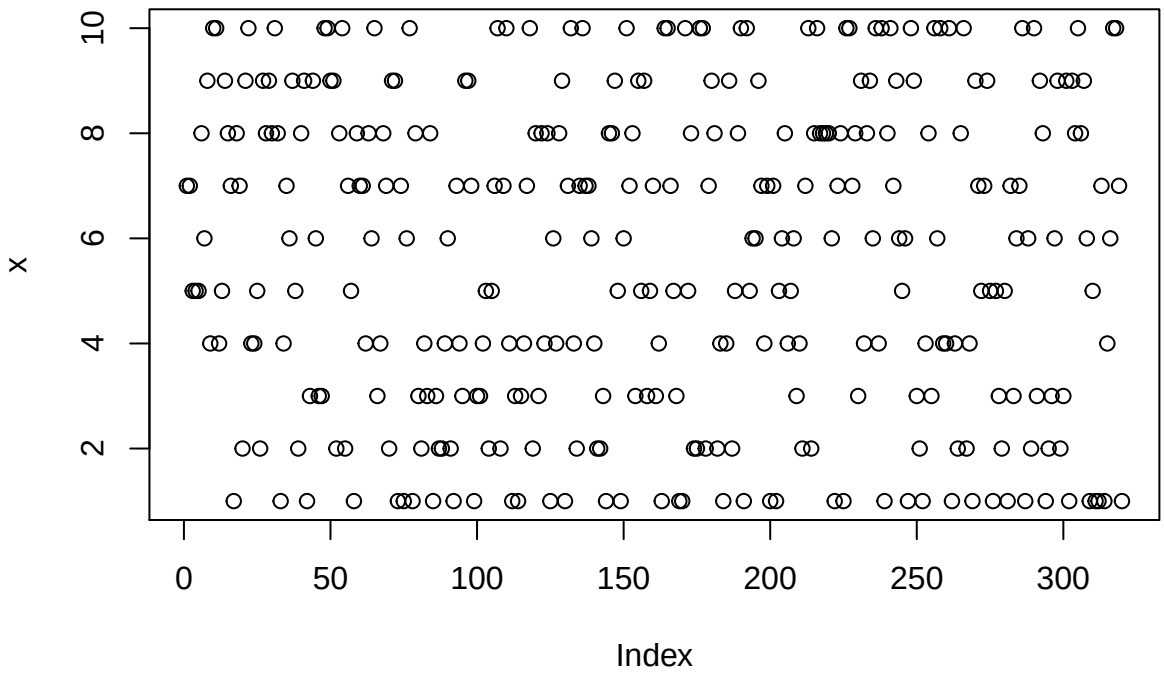
S3

S3 methods are very simple in definition and use. They are simply functions that are attached to `generic` functions to specify a `class` of object they are written for. When called they are chosen through a process called, "method dispatch". Here's an example with `plot`:

```
# Plotting numbers
x <- sample(1:10, 320, replace = TRUE)
typeof(x)
```

```
[1] "integer"
```

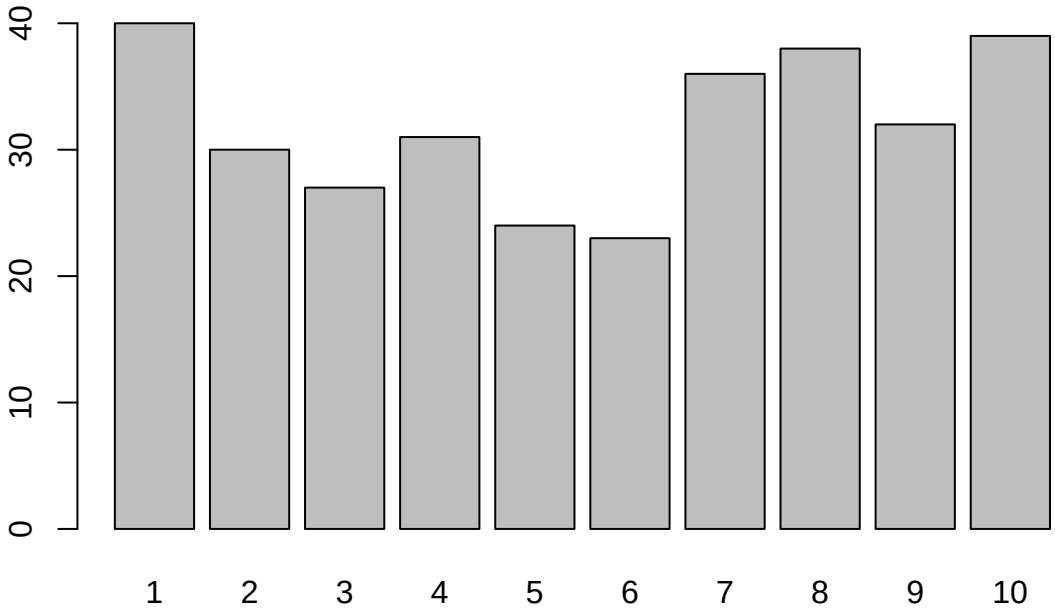
```
plot(x)
```




```
# Plotting factors
y <- factor(x)
typeof(y)
```

```
[1] "integer"
```

```
plot(y)
```



The first plot uses a function called `plot.default()`, while the second uses `plot.factor()`. These are both based on a default generic function, `plot`. A generic function is defined by generating a call to `UseMethod()`.

```
plot
```

```
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x7fe3830ef1b8>
<environment: namespace:graphics>
```

You can view all of the defined methods for a generic with `methods()`:

```
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.isoreg*        plot.lm*
[16] plot.medpolish*     plot.mlm*           plot.ppr*
[19] plot.prcomp*        plot.princomp*      plot.profile.nls*
[22] plot.raster*        plot.spec*          plot.stepfun
[25] plot.stl*           plot.table*         plot.ts
[28] plot.tskernel*      plot.TukeyHSD*
see '?methods' for accessing help and source code
```

You can also do the reverse and list generic functions for a particular class with `methods(class = "<class>")`:

```
methods(class = "factor")
```

```
 [1] [             [[            [[<-          [<-           all.equal
 [6] as.character  as.data.frame as.Date       as.list       as.logical
[11] as.POSIXlt    as.vector     coerce        droplevels    format
[16] initialize    is.na<-       length<-      levels<-      Math
[21] Ops           plot          print         relevel       relist
[26] rep           show          slotsFromS3   summary       Summary
```

```
[31] xtfrm
see '?methods' for accessing help and source code
```

S3 methods are defined by first defining the generic using `UseMethod`, then defining a set of class-specific methods as well as a default. Here is a generic to create a summary:

```
smrz <- function(x, ...) UseMethod("smrz")
smrz
```

```
function(x, ...) UseMethod("smrz")
```

```
str(smrz)
```

```
function (x, ...)
 - attr(*, "srcref")=Class 'srcref'  atomic [1:8] 1 9 1 42 9 42 1 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7fe3849746d0>
```

Here's a method for a `factor` vector:

```
smrz.factor <- function(x) {
  freq <- table(x)
  prop <- freq / sum(freq)
  cbind(freq = freq, prop = prop)
}

smrz(y)
```

```
   freq     prop
1    40 0.125000
2    30 0.093750
3    27 0.084375
4    31 0.096875
5    24 0.075000
6    23 0.071875
7    36 0.112500
8    38 0.118750
9    32 0.100000
10   39 0.121875
```

Let's use the same one for a `logical` vector:

```
smrz.logical <- function(x) smrz.factor(as.factor(x))

lg <- sample(c(T, F), 250, replace = T)
smrz(lg)
```

```
      freq  prop
FALSE  124 0.496
TRUE   126 0.504
```

Its good to define a `default` method for classes not explicitly defined:

```
smrz.default <- function(x) cat("Unknown class, can't summarize")
smrz(1:10)
```

```
Unknown class, can't summarize
```

You can create your own class by setting or adding to the `class` attribute of of an existing class. It is usually good to add to the class, because this will allow your new class to "inherit" features of the parent class. Functions designed to work on the parent class will continue to work on yours. Most of the time, S3 classes

are inherited from `list` objects, but any type of object is game. For example, we can create an object that is the result of a frequency summary:

```
smrz.factor <- function(x) {
  freq <- table(x)
  prop <- freq / sum(freq)
  result <- cbind(freq = freq, prop = prop)
  class(result) <- c(class(result), "factorSummary")
  result
}
```

Now we can create another method for a `factorSummary` object:

```
smrz.factorSummary <- function(x) {
  n = sum(x[, "freq"]) # Number of values
  H = -sum(x[, "prop"] * log(x[, "prop"])) # Shannon diversity index
  c(n = n, H = H)
}

# create a summary of a factor
y.smry <- smrz(y)
str(y.smry)
```

```
 matrix [1:10, 1:2] 40 30 27 31 24 23 36 38 32 39 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:10] "1" "2" "3" "4" ...
  ..$ : chr [1:2] "freq" "prop"
```

```
class(y.smry)
```

```
[1] "matrix"        "factorSummary"
```

```
# summarize the summary
y.smry.smry <- smrz(y.smry)
y.smry.smry
```

```
         n          H
320.000000   2.285698
```

S4

S4 methods and objects are more rigorously defined. In particular, S4 classes are formally defined objects with specific `slots` that can be set to have default values on creation. S4 objects use the `@` operator to access those slots. However, because they are so explicitly defined, it is better to create accessor functions that get and set data, rather than have users use `@`.

As an example we'll create an S4 class to contain data about a CTD station. This is a simple object that will have three slots: date, operator, and position.

To create an S4 class, you use the `setClass()` function. You have to supply a name for the class (`Class` argument), and name and define the `slots` in the class. For example, our CTD station class can be defined as:

```
setClass(
  Class = "ctdStation",
  slots = c(
```

```
    date = "character",
    operator = "character",
    position = "numeric"
  )
)
```

To create an object in this class, we can use the `new()` function:

```
st1 <- new("ctdStation", date = "01/05/2018", operator = "MKR", position = c(32.97, -117.39))
st1
```

```
An object of class "ctdStation"
Slot "date":
[1] "01/05/2018"

Slot "operator":
[1] "MKR"

Slot "position":
[1]   32.97 -117.39
```

Data is retrieved from slots in the object using either the `@` symbol (which is similar to `$`):

```
st1@date
```

```
[1] "01/05/2018"
```

. . . or the `slot()` function (which is similar to `[[]]`):

```
slot(st1, "operator")
```

```
[1] "MKR"
```

Constructor

Instead of using `new()`, it is usually better to create a "constructor" function for your S4 class. This often is just the name of the class. In the constructor, you should do validation checks.

```
ctdStation <- function(date, operator, position) {
  # check that operator is 2-3 characters long:
  if(!nchar(operator) %in% 2:3) {
    stop("operator must be 2 or 3 characters")
  }
  # convert operator to uppercase
  operator <- toupper(operator)
  new(
    "ctdStation",
    date = date,
    operator = operator,
    position = position
  )
}

ctdStation("04/23/2017", "eric", c(32.5, -117.3))
```

```
Error in ctdStation("04/23/2017", "eric", c(32.5, -117.3)): operator must be 2 or 3 characters
```

```
ctdStation("04/23/2017", "eia", c(32.5, -117.3))
```

```
An object of class "ctdStation"
Slot "date":
[1] "04/23/2017"

Slot "operator":
[1] "EIA"

Slot "position":
[1]   32.5 -117.3
```

Validity checks

You can also create a function to check the validity of an object (in case it isn't created with your constructor). Use `setValidity()` to return either TRUE if the object is good, or FALSE if it is not. Optionally, `setValidity` can also return a character string that describes what is wrong:

```
setValidity(
  Class = "ctdStation",
  method = function(object) {
    good.length <- nchar(object@operator) %in% 2:3
    is.upper <- toupper(object@operator) == object@operator
    good.length & is.upper
  }
)
```

```
Class "ctdStation" [in ".GlobalEnv"]

Slots:

Name:       date  operator  position
Class: character character   numeric
```

```
# This would create an invalid object, so it won't be created
st2 <- new("ctdStation", date = "01/05/2018", operator = "eric", position = c(32.97, -117.39))
```

```
Error in validObject(.Object): invalid class "ctdStation" object: FALSE
```

```
st2
```

```
Error in eval(expr, envir, enclos): object 'st2' not found
```

```
# while this one is good
st3 <- new("ctdStation", date = "01/05/2018", operator = "EIA", position = c(32.97, -117.39))
```

Methods

To create a method for an S4 object, use the `setMethod()` function. For example, we want to create a function that displays our CTD class. For S4 objects, the generic for this is `show`. We will create a method for `ctdStation`:

```
# The default `show` for ctdStation
st3
```

```
An object of class "ctdStation"
```

```
Slot "date":
[1] "01/05/2018"

Slot "operator":
[1] "EIA"

Slot "position":
[1]   32.97 -117.39
```

```
# Define a new method:
setMethod(
  f = "show",
  signature = "ctdStation",
  definition = function(object){
    cat("CTD Station", "\n")
    cat("-----------", "\n")
    cat("Date: ", object@date, "\n", sep = "")
    cat("Operator: ", object@operator, "\n", sep = "")
    cat("Position: ", object@position[1], ", ", object@position[2], "\n", sep = "")
  })
```

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

```
# Our new `show` method:
st3
```

```
CTD Station
-----------
Date: 01/05/2018
Operator: EIA
Position: 32.97, -117.39
```

We can create new generics with `setGeneric()`. These come in handy when creating "accessor" functions that are used to get and set data in slots in your class. These functions are usually the names of the slots:

```
setGeneric(
  name = "operator",
  def = function(x, ...) standardGeneric("operator")
)
```

```
[1] "operator"
```

Now we can create a method to extract the `@operator` slot:

```
setMethod(
  f = "operator",
  signature = "ctdStation",
  function(x, ...) x@operator
)
```

```
[1] "operator"
```

```
operator(st3)
```

```
[1] "EIA"
```

To create a method to set the `@operator` slot, we create a similar, but new generic and method:

```
setGeneric(
  name = "operator<-",
```

```
  def = function(x, value) standardGeneric("operator<-")
)
```

```
[1] "operator<-"
```

```
setMethod(
  f = "operator<-",
  signature = "ctdStation",
  function(x, value) {
    value <- toupper(value)
    x@operator <- value
    validObject(x) # check to make sure value doesn't invalidate the object
    x
  }
)
```

```
[1] "operator<-"
```

```
operator(st3) <- "myname"
```

```
Error in validObject(x): invalid class "ctdStation" object: FALSE
```

```
operator(st3) <- "abc"
st3
```

```
CTD Station
-----------
Date: 01/05/2018
Operator: ABC
Position: 32.97, -117.39
```

Inheritance

Classes can inherit from one another. This means that they contain the structures of their component classes and methods that work on one will work on the other. For example, we can make a new class that records the actual CTD data from a cast at a given station. Our new class will be based on the `ctdStation` class, but have a slot for a data.frame:

```
setClass(
  "ctdData",
  slots = c(cast = "data.frame"),
  contains = "ctdStation"
)

ctd1 <- new(
  "ctdData",
  date = "5/29/2017",
  operator = "EIA",
  position = c(39, -118),
  cast = data.frame(
    depth = c(1, 10, 100),
    temp = c(16, 12, 5),
    sal = c(33.2, 35, 37.9)
  )
)
```

```r
# Station info
ctd1
```

```
CTD Station
-----------
Date: 5/29/2017
Operator: EIA
Position: 39, -118
```

```r
# Cast data
ctd1@cast
```

```
  depth temp  sal
1     1   16 33.2
2    10   12 35.0
3   100    5 37.9
```

```r
# The `operator` method still works:
operator(ctd1) <- "XXX"
ctd1
```

```
CTD Station
-----------
Date: 5/29/2017
Operator: XXX
Position: 39, -118
```