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()`.

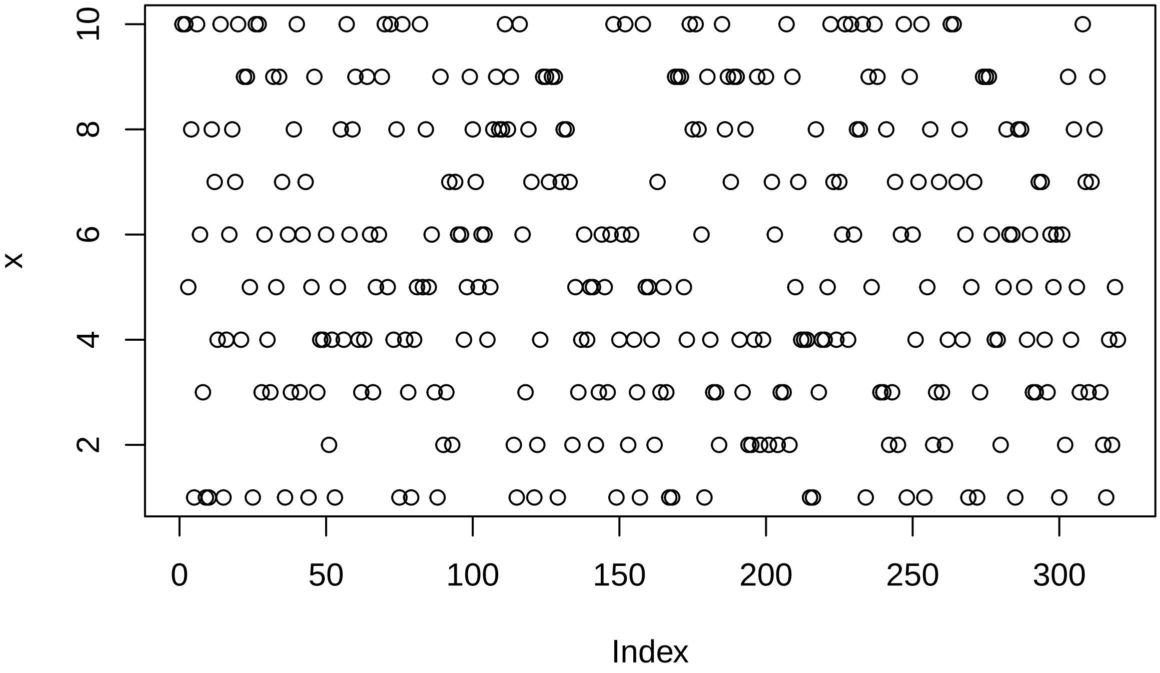
S3

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"
```

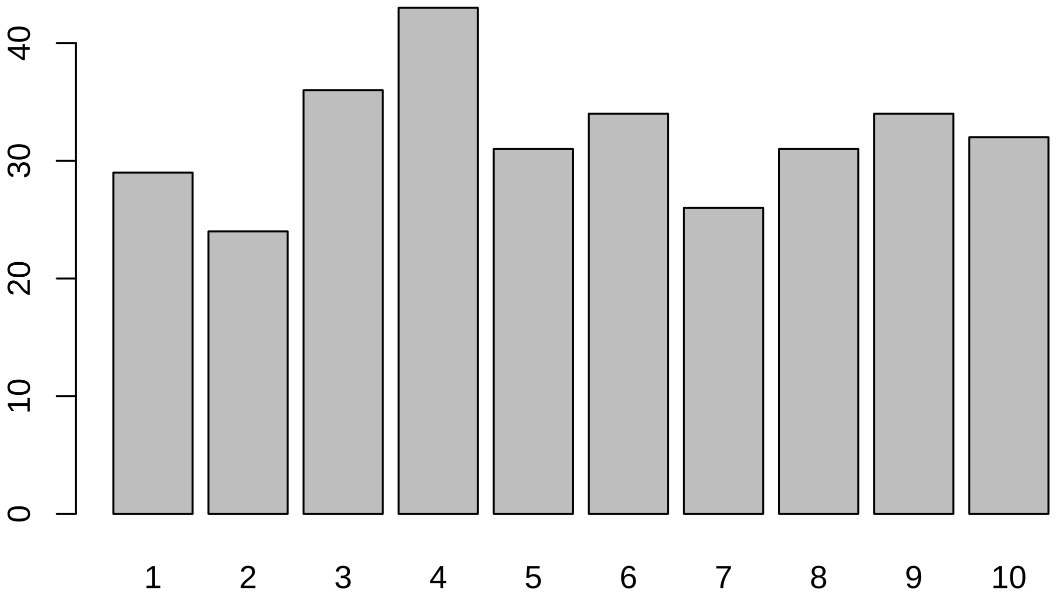
```
plot(x)
```



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

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

```r
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()`.

```r
plot
```

```
function (x, y, ...) 
UseMethod("plot")
<bytecode: 0x7f83572513b8>
<environment: namespace:graphics>
```

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

```r
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>")`:

```r
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: 0x7f8359cc5028>
```

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    29 0.090625
2    24 0.075000
3    36 0.112500
4    43 0.134375
5    31 0.096875
6    34 0.106250
7    26 0.081250
8    31 0.096875
9    34 0.106250
10   32 0.100000
```

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  111 0.444
TRUE   139 0.556
```

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 adding to the `class` attribute of of an existing class. Most of the time, this is done to `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] 29 24 36 43 31 34 26 31 34 32 ...
 - 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   2.290268
```

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 also use the `@` operator to access those slots. However, because they are so explicitly defined, it is more common 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 from a CTD cast: