Functions

Function Parts

Functions are defined by two components: the arguments (`formals`) and the code (`body`). Functions are assigned names like any other object in R (using `=` or `<-`)

```
gcd = function(long1, lat1, long2, lat2) {
  R = 6371 # Earth mean radius in km

  # distance in km
  acos(sin(lat1)*sin(lat2) + cos(lat1)*cos(lat2) * cos(long2-long1)) * R
}
```

Returning values

There are two approaches to returning values from functions in R - explicit and implicit return values.

Explicit - using one or more `return` statements

```
f = function(x) {
  x + 1
  return(x * x)
}
f(2)
```

```
## [1] 4
```

Implicit - return value of the last expression is returned.

```
g = function(x) {
  x + 1
  x * x
}
g(3)
```

```
## [1] 9
```

Returning multiple values

If we want a function to return more than one value we can group things using either vectors or lists.

```
f = function(x) {
  c(x, x^2, x^3)
}

f(1:2)
```

```
## [1] 1 2 1 4 1 8
```

```
g = function(x) {
  list(x, "hello")
}

g(1:2)
```

```
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] "hello"
```

Argument names

When defining a function we are also implicitly defining names for the arguments, when calling the function we can use these names to pass arguments in a alternative order.

```
f = function(x, y, z) {
  paste0("x=", x, " y=", y, " z=", z)
}
```

```
f(1, 2, 3)
```

```
## [1] "x=1 y=2 z=3"
```

```
f(y=2, 1, 3)
```

```
## [1] "x=1 y=2 z=3"
```

```
f(z=1, x=2, y=3)
```

```
## [1] "x=2 y=3 z=1"
```

```
f(y=2, 1, x=3)
```

```
## [1] "x=3 y=2 z=1"
```

```
f(1, 2, 3, 4)
```

```
## Error in f(1, 2, 3, 4): unused argument (4)
```

```
f(1, 2, m=3)
```

```
## Error in f(1, 2, m = 3): unused argument (m = 3)
```

Argument defaults

It is also possible to give function arguments default values, so that they don't need to be provided every time the function is called.

```
f = function(x, y=1, z=1) {
  paste0("x=", x, " y=", y, " z=", z)
}
```

```
f(3)
```

```
## [1] "x=3 y=1 z=1"
```

```
f(z=3, x=2)
```

```
## [1] "x=2 y=1 z=3"
```

```
f(x=3)
```

```
## [1] "x=3 y=1 z=1"
```

```
f(y=2, 2)
```

```
## [1] "x=2 y=2 z=1"
```

```
f()
```

```
## Error in paste0("x=", x, " y=", y, " z=", z): argument "x" is missing, with no default
```

Scope

R has generous scoping rules, if it can't find a variable in the functions body, it will look for it in the next higher scope, and so on.

```
y = 1
f = function(x) {
  x + y
}

f(3)
```

```
## [1] 4
```

```
y = 1
g = function(x) {
  y = 2
  x + y
}

g(3)
```

```
## [1] 5
```

Additionally, variables defined within a scope only persist for the duration of that scope, and do not overwrite variables at a higher scopes

```
x = 1
y = 1
z = 1
f = function() {
    y = 2
    g = function() {
      z = 3
      return(x + y + z)
    }
    return(g())
}
f()
```

```
## [1] 6
```

```
c(x,y,z)
```

```
## [1] 1 1 1
```

Exercise 1 - scope

What is the output of the following code? Explain why.

```
z = 1

f = function(x, y, z) {
  z = x+y

  g = function(m = x, n = y) {
    m/z + n/z
  }

  z * g()
}

f(1, 2, x = 3)
```

Operators as functions

In R, operators are actually a special type of function - using backticks around the operator we can write them as functions.

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

```
typeof(`+`)
```

```
## [1] "builtin"
```

```
x = 4:1
x + 2
```

```
## [1] 6 5 4 3
```

```
`+`(x, 2)
```

```
## [1] 6 5 4 3
```

Getting Help

Prefixing any function name with a `?` will open the related help file for that function.

```
?`+`
?sum
```

For functions not in the base package, you can generally see their implementation by entering the function name without parentheses (or using the `body` function).

```
lm
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## {
##     ret.x <- x
##     ret.y <- y
##     cl <- match.call()
##     mf <- match.call(expand.dots = FALSE)
##     m <- match(c("formula", "data", "subset", "weights", "na.action",
##         "offset"), names(mf), 0L)
##     mf <- mf[c(1L, m)]
##     mf$drop.unused.levels <- TRUE
##     mf[[1L]] <- quote(stats::model.frame)
##     mf <- eval(mf, parent.frame())
##     if (method == "model.frame")
##         return(mf)
```

Attributes

Attributes

Attributes are metadata that can be attached to objects in R. Some are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, etc.) and change the way in which an object behaves in R.

Attributes are implemented as a named list that is attached to an object. They can be interacted with via the `attr` and `attributes` functions.

```
(x = c(L=1,M=2,N=3))
```

```
## L M N
## 1 2 3
```

```
attributes(x)
```

```
## $names
## [1] "L" "M" "N"
```

```
str(attributes(x))
```

```
## List of 1
##  $ names: chr [1:3] "L" "M" "N"
```

```r
attr(x, "names") = c("A","B","C")
x
```

```
## A B C
## 1 2 3
```

```r
names(x)
```

```
## [1] "A" "B" "C"
```

```r
names(x) = c("Z","Y","X")
x
```

```
## Z Y X
## 1 2 3
```

```r
names(x) = 1:3
x
```

```
## 1 2 3
## 1 2 3
```

```r
attributes(x)
```

```
## $names
## [1] "1" "2" "3"
```

```r
names(x) = c(TRUE, FALSE, TRUE)
x
```

```
##  TRUE FALSE  TRUE
##     1     2     3
```

```r
attributes(x)
```

```
## $names
## [1] "TRUE"  "FALSE" "TRUE"
```

Factors

Factor objects are how R represents categorical data (e.g. a variable where there are a fixed # of possible outcomes).

```
(x = factor(c("Sunny", "Cloudy", "Rainy", "Cloudy", "Cloudy")))
```

```
## [1] Sunny  Cloudy Rainy  Cloudy Cloudy
## Levels: Cloudy Rainy Sunny
```

```
str(x)
```

```
##  Factor w/ 3 levels "Cloudy","Rainy",..: 3 1 2 1 1
```

```
typeof(x)
```

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

Composition

A factor is just an integer vector with two attributes: `class = "factor"` and `levels`.

```
x
```

```
## [1] Sunny  Cloudy Rainy  Cloudy Cloudy
## Levels: Cloudy Rainy Sunny
```

```
attributes(x)
```

```
## $levels
## [1] "Cloudy" "Rainy"  "Sunny"
##
## $class
## [1] "factor"
```

We can build our own factor from scratch using,

```
y = c(3L, 1L, 2L, 1L, 1L)
attr(y, "levels") = c("Cloudy", "Rainy", "Sunny")
attr(y, "class") = "factor"
y
```

```
## [1] Sunny  Cloudy Rainy  Cloudy Cloudy
## Levels: Cloudy Rainy Sunny
```

S3 Object System

`class`

The `class` attribute is an additional layer to R's type hierarchy,

value	`typeof()`	`mode()`	`class()`
`TRUE`	logical	logical	logical
`1`	double	numeric	numeric
`1L`	integer	numeric	integer
`"A"`	character	character	character
`NULL`	NULL	NULL	NULL
`list(1, "A")`	list	list	list
`factor("A")`	integer	numeric	factor
`function(x) x^2`	closure	function	function

S3 class specialization

```
x = c("A","B","A","C")
print( x )
```

```
## [1] "A" "B" "A" "C"
```

```
print( factor(x) )
```

```
## [1] A B A C
## Levels: A B C
```

```
print( unclass( factor(x) ) )
```

```
## [1] 1 2 1 3
## attr(,"levels")
## [1] "A" "B" "C"
```

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x7fef522faee0>
## <environment: namespace:base>
```

Other examples

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fef51ee21d0>
## <environment: namespace:base>
```

```
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x7fef5283c940>
## <environment: namespace:base>
```

```
t.test
```

```
## function (x, ...)
## UseMethod("t.test")
## <bytecode: 0x7fef5381e620>
## <environment: namespace:stats>
```

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fef539312c0>
## <environment: namespace:base>
```

Not all base functions use this approach,

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

What is S3?

> S3 is R's first and simplest OO system. It is the only OO system used in the base and stats packages, and it's the most commonly used system in CRAN packages. S3 is informal and ad hoc, but it has a certain elegance in its minimalism: you can't take away any part of it and still have a useful OO system.
>
> — Hadley Wickham, Advanced R

- S3 should not be confused with R's other object oriented systems:
  S4, Reference classes, and R6*.

What's going on?

S3 objects and their related functions work using a very simple dispatch mechanism - a generic function is created whose sole job is to call the `UseMethod` function which then calls a class specialized function using the naming convention: `generic.class`.

We can see all of the specialized versions of the generic using the `methods` function.

```
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
```

```
methods("print")
```

```
##   [1] print.acf*
##   [2] print.AES*
##   [3] print.anova*
##   [4] print.aov*
##   [5] print.aovlist*
##   [6] print.ar*
##   [7] print.Arima*
##   [8] print.arima0*
##   [9] print.AsIs
##  [10] print.aspell*
##  [11] print.aspell_inspect_context*
##  [12] print.bibentry*
##  [13] print.Bibtex*
##  [14] print.browseVignettes*
##  [15] print.by
##  [16] print.changedFiles*
##  [17] print.check_code_usage_in_package*
##  [18] print.check_compiled_code*
##  [19] print.check_demo_index*
##  [20] print.check_depdef*
##  [21] print.check_details*
##  [22] print.check_details_changes*
##  [23] print.check_doi_db*
##  [24] print.check_dotInternal*
##  [25] print.check_make_vars*
##  [26] print.check_nonAPI_calls*
##  [27] print.check_package_code_assign_to_globalenv*
##  [28] print.check_package_code_attach*
```

```
print.factor
```

```
## function (x, quote = FALSE, max.levels = NULL, width = getOption("width"),
##     ...)
## {
##     ord <- is.ordered(x)
##     if (length(x) == 0L)
##         cat(if (ord)
##             "ordered"
##         else "factor", "(0)\n", sep = "")
##     else {
##         xx <- character(length(x))
##         xx[] <- as.character(x)
##         keepAttrs <- setdiff(names(attributes(x)), c("levels",
##             "class"))
##         attributes(xx)[keepAttrs] <- attributes(x)[keepAttrs]
##         print(xx, quote = quote, ...)
##     }
##     maxl <- if (is.null(max.levels))
##         TRUE
##     else max.levels
##     if (maxl) {
##         n <- length(lev <- encodeString(levels(x), quote = ifelse(quote,
##             "\"", "")))
##         colsep <- if (ord)
##             " < "
##         else " "
##         T0 <- "Levels: "
##         if (is.logical(maxl))
##             maxl <- {
```

```
print.integer
```

```
## Error in eval(expr, envir, enclos): object 'print.integer' not found
```

```
print.default
```

```
## function (x, digits = NULL, quote = TRUE, na.print = NULL, print.gap = NULL,
##     right = FALSE, max = NULL, width = NULL, useSource = TRUE,
##     ...)
## {
##     args <- pairlist(digits = digits, quote = quote, na.print = na.print,
##         print.gap = print.gap, right = right, max = max, width = width,
##         useSource = useSource, ...)
##     missings <- c(missing(digits), missing(quote), missing(na.print),
##         missing(print.gap), missing(right), missing(max), missing(width),
##         missing(useSource))
##     .Internal(print.default(x, args, missings))
## }
## <bytecode: 0x7fef54b1d110>
## <environment: namespace:base>
```

The other way

If instead we have a class and want to know what specialized functions exist for that class, then we can again use the `methods` function with the `class` argument.

```
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
```

Adding methods

```
x = structure(c(1,2,3), class="class_A")
x
```

```
## [1] 1 2 3
## attr(,"class")
## [1] "class_A"
```

```
print.class_A = function(x) {
  cat("Class A!\n")
  print.default(unclass(x))
}

x
```

```
## Class A!
## [1] 1 2 3
```

```
class(x) = "class_B"
x
```

```
## Class B!
## [1] 1 2 3
```

```
y = structure(c(6,5,4), class="class_B")
y
```

```
## [1] 6 5 4
## attr(,"class")
## [1] "class_B"
```

```
print.class_B = function(x) {
  cat("Class B!\n")
  print.default(unclass(x))
}

y
```

```
## Class B!
## [1] 6 5 4
```

```
class(y) = "class_A"
y
```

```
## Class A!
## [1] 6 5 4
```

Defining a new S3 Generic

```
shuffle = function(x) {
  UseMethod("shuffle")
}
```

```
shuffle.default = function(x) {
  stop("Class ", class(x), " is not supported by shuffle.\n", call. = FALSE)
}
```

```
shuffle.factor = function(f) {
  factor( sample(as.character(f)), levels = sample(levels(f)) )
}

shuffle.integer = function(x) {
  sample(x)
}
```

```
shuffle( 1:10 )
```

```
##  [1] 10  7  8  1  5  6  9  3  2  4
```

```
shuffle( factor(c("A","B","C","A")) )
```

```
## [1] C A A B
## Levels: B A C
```

```
shuffle( c(1, 2, 3, 4, 5) )
```

```
## Error: Class numeric is not supported by shuffle.
```

```
shuffle( letters[1:5] )
```

```
## Error: Class character is not supported by shuffl
```

Exercise 2 - classes, modes, and types

Below we have defined an S3 method called `report`, it is designed to return a message about the type/mode/class of an object passed to it.

```
report = function(x) {
  UseMethod("report")
}

report.default = function(x) {
  "This class does not have a method defined."
}
```

```
report.integer = function(x) {
 "I'm an integer!"
}

report.double = function(x) {
  "I'm a double!"
}

report.numeric = function(x) {
  "I'm a numeric!"
}
```

Try running the `report` function with different input types, what happens?
Now run `rm("report.integer")` in your console and try using the `report` function again, what has changed? What does this tell us about S3, types, modes, and classes?

Acknowledgments

Above materials are derived in part from the following sources:

- Hadley Wickham - Advanced R
- R Language Definition