Functions

FRE6871 & FRE7241, Spring 2023

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

May 12, 2023

NYU | TANDON SCHOOL OF ENGINEERING

Functions in R

R functions have three components:

- a list of formal arguments,
- a body containing R code,
- an environment,

An R function plus its environment is referred to as a function *closures*.

The function body should be enclosed in curly braces {}, unless it contains a single command, then it doesn't have to enclosed.

The function body doesn't require a `return` statement, since by default R functions return the last statement evaluated in the body.

`args()` displays the formal arguments of a function.

```
> # Define a function with two arguments
> testfun <- function(first_arg, second_arg) {  # Body
+   first_arg + second_arg  # Returns last evaluated statement
+ }  # end testfun
>
> testfun(1, 2)  # Apply the function
> args(testfun)  # Display argument
>
> # Define function that uses variable from enclosure environment
> testfun <- function(first_arg, second_arg) {
+   first_arg + second_arg + globv
+ }  # end testfun
>
> testfun(3, 2)  # error - globv doesn't exist yet!
> globv <- 10  # Create globv
> testfun(3, 2)  # Now works
```

Return Values of Functions

The function body doesn't require a `return` statement, since by default `R` functions return the last statement evaluated in the body.

`return()` statements are inserted in logical branches to terminate function execution and return its intended value.

```
> # Define function that returns NULL for non-numeric argument
> testfun <- function(input) {
+   if (!is.numeric(input)) {
+     warning(paste("argument", input, "isn't numeric"))
+     return(NULL)
+   }
+   2*input
+ }  # end testfun
>
> testfun(2)
> testfun("hello")
```

Functions That Return `invisible`

If a return value is wrapped in the function `invisible()` then the return value isn't printed.

But if the function is assigned to a variable, then its return value is assigned to that variable.

`invisible()` allows creating functions whose return values can be assigned, but which do not print when they're not assigned.

The function `load()` reads data from `.RData` files, and *invisibly* returns a vector of names of objects created in the workspace.

```
> # Define a function that returns invisibly
> return_invisible <- function(input) {
+   invisible(input)
+ }  # end return_invisible
>
> return_invisible(2)
>
> globv <- return_invisible(2)
> globv
>
> rm(list=ls())  # Remove all objects
> # Load objects from file
> loaded <- load(file="/Users/jerzy/Develop/data/my_data.RData")
> loaded  # Vector of loaded objects
> ls()  # List objects
```

Binding Function Arguments

The formal arguments of a function are defined in its argument list.

When a function is called, it's passed a list of actual function arguments.

Formal arguments can be *bound* to actual arguments either by name or by position:

- by name: formal arguments are *bound* to actual arguments with the same name,
- by position: the first formal argument is *bound* to the first actual argument, etc.

Binding by name takes precedence over *binding* by position: first all the named arguments are *bound*, then the remaining arguments are *bound* by position.

Partial argument names are *bound* to full names.

```
> testfun <- function(first_arg, second_arg) {
+ # Last statement of function is return value
+   first_arg + 2*second_arg
+ }  # end testfun
> testfun(first_arg=3, second_arg=2)  # Bind by name
> testfun(first=3, second=2)  # Partial name binding
> testfun(3, 2)  # Bind by position
> testfun(second_arg=2, 3)  # mixed binding
> testfun(3, 2, 1)  # Too many arguments
> testfun(2)  # Not enough arguments
```

All the actual arguments must be *bound* to formal arguments, and if not then an `"unused argument"` error is produced.

If there aren't enough formal arguments, then an `"argument is missing"` error is produced,

Default Values for Arguments

Formal arguments may be assigned default values, so that when the actual arguments are missing then their default values are used instead.

Default values are often assigned to function parameters, that determine the function's behavior.

Default values can be specified as a vector of strings, representing the possible values of a function's parameter.

The function `match.arg()` matches a string to one of the possible values, and returns the matched value, or produces an `error` if it can't match it.

The function `str()` displays the structure of an `R` object, for example a function name and its formal arguments.

```
> # Function "paste" has two arguments with default values
> str(paste)
> # Default values of arguments can be specified in argument list
> testfun <- function(first_arg, ratio=1) {
+   ratio*first_arg
+ }  # end testfun
> testfun(3)  # Default value used for second argument
> testfun(3, 2)  # Default value over-ridden
> # Default values can be a vector of strings
> testfun <- function(input=c("first_val", "second_val")) {
+   input <- match.arg(input)  # Match to arg list
+   input
+ }  # end testfun
> testfun("second_val")
> testfun("se")  # Partial name binding
> testfun("some_val")  # Invalid string
```

Function for Calculating Skew

R provides an easy way for users to write functions.

Formal function arguments can be bound to input variables by position or by name.

If the function arguments are missing then their default value is used.

Functions return the value of the last expression that is evaluated.

datasets is a base package containing various datasets, for example: EuStockMarkets.

The EuStockMarkets dataset contains daily closing prices of european stock indices.

```
> # VTI percentage returns
> retp <- rutils::diffit(log(Cl(rutils::etfenv$VTI)))
> # calc_skew() calculates skew of time series of returns
> # Default is normal time series
> calc_skew <- function(retp=rnorm(1000)) {
+   # Number of observations
+   nrows <- NROW(retp)
+   # Standardize returns
+   retp <- (retp - mean(retp))/sd(retp)
+   # Calculate skew - last statement automatically returned
+   nrows*sum(retp^3)/((nrows-1)*(nrows-2))
+ }  # end calc_skew
>
> # Calculate skew of DAX returns
> # Bind arguments by name
> calc_skew(retp=retp)
> # Bind arguments by position
> calc_skew(retp)
> # Use default value of arguments
> calc_skew()
```

The dots "..." Function Argument

The dots "..." function argument is a formal argument without a name, as opposed to the other formal arguments which all have names.

The dots "..." bind with any number of additional arguments, that aren't already bound by name or position to the named arguments.

The dots "..." are used when the number of arguments isn't known in advance, and allows functions to accept an indefinite number of arguments.

The dots "..." are sometimes placed *after* the named arguments, to allow passing of additional parameters into a function.

Functionals often place the dots "..." argument *after* the named arguments, to allow passing the dots "..." to the function being called by the *functional*.

```
> str(plot)  # Dots for additional plot parameters
> bind_dots <- function(input, ...) {
+   paste0("input=", input, ", dots=", paste(..., sep=", "))
+ }  # end bind_dots
> bind_dots(1, 2, 3)  # "input" bound by position
> bind_dots(2, input=1, 3)  # "input" bound by name
> bind_dots(1, 2, 3, foo=10)  # Named argument bound to dots
> bind_dots <- function(arg1, arg2, ...) {
+   arg1 + 2*arg2 + sum(...)
+ }  # end bind_dots
> bind_dots(3, 2)  # Bind arguments by position
> bind_dots(3, 2, 5, 8)  # Extra arguments bound to dots
```

Argument Binding With dots "..." Argument

The dots "..." argument is sometimes placed *before* the named arguments, so that a function can accept an indefinite number of arguments, without binding them by position with the named arguments.

When the dots "..." are placed *before* the named arguments, the named arguments are often assigned default values, so they don't have to be bound to a value in the call.

Arguments that appear after the dots "..." must be *bound* by their full name, and can't be partially *bound*.

```
> str(sum)  # Dots before other arguments
> sum(1, 2, 3)  # Dots bind before other arguments
> sum(1, 2, NA, 3, na.rm=TRUE)
> bind_dots <- function(..., input) {
+   paste0("input=", input,
+  ", dots=", paste(..., sep=", "))
+ }  # end bind_dots
> # Arguments after dots must be bound by full name
> bind_dots(1, 2, 3, input=10)
> bind_dots(1, 2, 3, input=10, foo=4)  # Dots bound
> bind_dots(1, 2, 3)  # "input" not bound
> bind_dots <- function(..., input=10) {
+   paste0("input=", input,
+  ", dots=", paste(..., sep=", "))
+ }  # end bind_dots
> bind_dots(1, 2, 3)  # "input" not bound, but has default
```

Wrapper Functions With dots "..." Argument

Wrapper functions provide a convenient user interface to functions, by assigning default argument values, validating data, and formatting the output.

Wrapper functions are designed to perform the actions of other functions, while reducing their complexity.

The dots "..." argument of the *wrapper* function allows passing additional arguments on to the wrapped function.

Wrapper functions should be used with caution, since wrapping a function creates extra code (overhead), which slows down R.

```
> # Wrapper for mean() with default na.rm=TRUE
> my_mean <- function(x, na.rm=TRUE, ...) {
+   mean(x=x, na.rm=na.rm, ...)
+ }  # end my_mean
> foo <- sample(c(1:10, NA, rep(0.1, t=5)))
> mean(c(foo, NA))
> mean(c(foo, NA), na.rm=TRUE)
> my_mean(c(foo, NA))
> my_mean(c(foo, NA), trim=0.4)  # Pass extra argument
> # Wrapper for saving data into default directory
> save_data <- function(...,
+               file=stop("error: no file name"),
+               my_dir="/Users/jerzy/Develop/data") {
+ # Create file path
+   file <- file.path(my_dir, file)
+   save(..., file=file)
+ }  # end save_data
> foo <- 1:10
> save_data(foo, file="scratch.RData")
> save_data(foo, file="scratch.RData", my_dir="/Users/jerzy/Develop"
> # Wrapper for testing negative arguments
> stop_if_neg <- function(input) {
+   if (!is.numeric(input) || input<0)
+     stop("argument not numeric or negative")
+ }  # end stop_if_neg
> # Wrapper for sqrt()
> my_sqrt <- function(input) {
+   stop_if_neg(input)
+   sqrt(input)
+ }  # end my_sqrt
> my_sqrt(2)
> my_sqrt(-2)
> my_sqrt(NA)
```

Recursive Functions with dots "..." Argument

Recursive functions can also accept the dots "..." argument.

The dots "..." argument can be referenced inside a function by first converting it into a list using "list(...)".

The function missing() returns TRUE if an argument is missing, and FALSE otherwise.

```
> # Recursive function sums its argument list
> sum_dots <- function(input, ...) {
+   if (missing(...)) {  # Check if dots are empty
+     return(input)  # just one argument left
+   } else {
+     input + sum_dots(...)  # Sum remaining arguments
+   }  # end if
+ }  # end sum_dots
> sum_dots(1, 2, 3, 4)
> # Recursive function sums its argument list
> sum_dots <- function(input, ...) {
+   if (NROW(list(...)) == 0) {  # Check if dots are empty
+     return(input)  # just one argument left
+   } else {
+     input + sum_dots(...)  # Sum remaining arguments
+   }  # end if
+ }  # end sum_dots
> sum_dots(1, 2, 3, 4)
```

Recursive Function for Calculating Fibonacci Sequence

Recursive functions call themselves in their own body.

The *Fibonacci* sequence of integers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

$$F_1 = 0, F_2 = 1,$$

$$F_n = 0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

The *Fibonacci* sequence was invented by *Indian* mathematicians, and later described by the Italian mathematician *Fibonacci* in his famous treatise *Liber Abaci*.

```
> fibonacci <- function(nrows) {
+   if (nrows > 2) {
+     fib_seq <- fibonacci(nrows-1)  # Recursion
+     c(fib_seq, sum(tail(fib_seq, 2)))  # Return this
+   } else {
+     c(0, 1)  # Initialize and return
+   }
+ }  # end fibonacci
> fibonacci(10)
> tail(fibonacci(9), 2)
```

Exploring Functions

If a function name is called alone without arguments, then R displays the function code (but it must be on the search path).

Non-visible objects can't be viewed by calling their name.

The function `getAnywhere()` displays information about R objects, including non-visible objects.

The function `getAnywhere()` also displays R objects that aren't on the search path.

```
> # Show the function code
> plot.default
> # Display function
> getAnywhere(plot.default)
```

Internal and Primitive Functions

R is a high-confl language written in lower-confl languages, mostly C++ and some Fortran.

R functions are either written in R code (*interpreted* functions), or they directly call compiled C++ or Fortran code (*compiled* functions, also called *internal* or *primitive*).

R parses the code of *interpreted* functions, and eventually calls compiled C++ or Fortran code.

But this extra processing makes *interpreted* functions much slower than *compiled* functions.

Users can distinguish between *interpreted* functions and *compiled* functions by typing their names, and analyzing their source code.

The source code of *interpreted* functions contains multiple lines of R code, or a call to function `UseMethod()` (which *dispatches methods* associated with *generic* functions).

The source code of *compiled* functions contains a single call to one of the functions that execute *compiled* C++ or Fortran code: `.Internal()`, `.Primitive()`, `.C()`, `.Call()`, `.Fortran()`, or `.External()`.

```
> # Sum() is a compiled primitive function
> sum
> # mean() is a generic function
> mean
> # Show all methods of mean()
> methods(generic.function=mean)
> # Show code for mean.default()
> mean.default
```

Exploring Internal and Primitive Functions

Several functions call compiled code: `.C()`, `.Call()`, `.Fortran()`, `.External()`, or `.Internal()` and `.Primitive()`

`R .Internal() .Primitive()`

The function `getAnywhere()` displays `R` objects, including functions.

If a function name is called alone then `R` displays the function code (but it must be on the search path).

the user can access symbols from a package that isn't attached using the double-colon operator
`tools::file_ext`
The function `getAnywhere()` also displays `R` objects that aren't on the search path.

```
> # Get all methods for generic function "plot"
> methods("plot")
>
> getAnywhere(plot)  # Display function
```

Lazy Evaluation of Function Arguments

R functions delay evaluation of their arguments until they're needed by their R code.

This is called *lazy* evaluation.

If the function body doesn't evaluate an argument, then the function won't produce an error, even if the argument is missing.

```
> lazyfun <- function(arg1, arg2) {  # Define function lazyfun
+   2*arg1  # just multiply first argument
+ }  # end lazyfun
> lazyfun(3, 2)  # Bind arguments by position
> lazyfun(3)  # Second argument was never evaluated!
> lazyfun <- function(arg1, arg2) {  # Define function lazyfun
+   cat(arg1, '\n')  # Write to output
+   cat(arg2)  # Write to output
+ }  # end lazyfun
> lazyfun(3, 2)  # Bind arguments by position
> lazyfun(3)  # First argument written to output
```

Function Environments

When a function is called, a new *evaluation* environment is created.

The *evaluation* environment contains the function arguments and locally defined variables.

R evaluates variables inside functions by searching first in the *evaluation* environment, then the *enclosure* environment, then the R search path.

The enclosure of the *evaluation* environment is the environment where the function was defined.

The enclosure of functions defined in the workspace is the *global* environment.

The enclosure of functions defined in packages is the package *namespace*.

Objects defined in the function enclosure can be referenced inside the function.

```
> globv <- 1  # Define a global variable
> ls(environment())  # Get all variables in environment
> func_env <- function() {  # Explore function environments
+   locvar <- 1  # Define a local variable
+   cat('objects in evaluation environment:\t',
+       ls(environment()), '\n')
+   cat('objects in enclosing environment:\t',
+       ls(parent.env(environment())), '\n')
+   cat('this is the enclosing environment:')
+   parent.env(environment())  # Return enclosing environment
+ }  # end func_env
> func_env()
>
> environment(func_env)
> environment(print)  # Package namespace is the enclosure
```

Lexical Function Scope

A *free* variable is a variable that's not included in the *evaluation* environment.

Scoping rules determine how *free* variables are evaluated.

By default R uses *lexical* (*static*) scoping, which means that variables are first evaluated in the *evaluation* environment, then in the *enclosing* environment in which the function was *defined*, and so on.

Dynamic scoping means that variables are evaluated in the environment from which the function was *called*.

The standard assignment operator "<-" modifies variables in the *evaluation* environment.

The super-assignment operator "<<-" modifies variables in the *enclosing* environment.

```
> globv <- 1  # Define a global variable
> probe_scope <- function() {  # Explore function scope
+   locvar <- 2*globv  # Define a local variable
+   new_globvar <<- 11  # Define a global variable
+   cat('objects in evaluation environment:\t',
+       ls(environment()), '\n')
+   cat('this is a local locvar:\t', locvar, '\n')
+   cat('objects in enclosing environment:\n',
+       ls(parent.env(environment())), '\n')
+   cat('this is globv:\t', globv, '\n')
+   globv <- 10  # Define local globv
+   cat('this is the local globv:\t', globv, '\n')
+ }  # end probe_scope
> probe_scope()
> globv  # Global variable is unaffected
> new_globvar  # new_globvar is preserved
> locvar  # Local variable is gone!
```

Argument Passing in R

In general, arguments can be passed into functions either by *value* or by *reference*.

When an argument is passed by *value*, then a copy of that argument is passed to the function.

That way if the function modifies that argument, then the original object isn't modified.

When an argument is passed by *reference*, then a *pointer* to the original object is passed to the function.

If the function modifies that argument, then the original object is modified as well.

R uses a hybrid method of argument passing called *copy-on-modify semantics*.

R passes arguments by reference, thus saving memory space and time for copying.

But if the argument is modified within the function, then R makes a copy of it, so that the original object is unchanged.

```
> a <- 1  # Define a variable
> # New variable "b" points to value of "a"
> b <- a  # Define a new variable
> # When "b" is modified, R makes a copy of it
> b <- b+1
> # Function doubles its argument and returns it
> double_it <- function(input) {
+   input <- 2*input
+   cat("input argument was doubled to:", input, "\n")
+   input
+ }
> double_it(a)
> a  # variable "a" is unchanged
```

Copy-on-modify semantics has important implications for performance and memory usage.

http://stackoverflow.com/questions/15759117/what-exactly-is-copy-on-modify-semantics-in-r-and-where-is-the-c

Side effects Using the Super-assignment Operator "<<-"

Function *side effects* are operations on objects outside a function's *evaluation* environment.

The functions `plot()` and `load()` are examples of functions that produce *side effects*.

`load()` reads data from an `.RData` file, and creates objects in the workspace that are contained in the `.RData` file.

The super-assignment operator `"<<-"` allows creating functions that produce *side effects*.

The super-assignment operator `"<<-"` modifies or creates variables in the *enclosing* environment in which a function was *defined* (*lexical* scoping).

If a function was *defined* in the *global* environment then that's the function's *enclosing* environment, and the `"<<-"` operator operates on variables in the *global* environment.

```
> rm(list=ls())  # Remove all objects
> ls()  # List objects
> # Load objects from file (side effect)
> load(file="my_data.RData")
> ls()  # List objects
> globv <- 1  # Define a global variable
> # Explore function scope and side effects
> side_effect <- function() {
+   cat("global globv =", globv, "\n")
+ # Define local "globv" variable
+   globv <- 10
+   cat("local globv =", globv, "\n")
+   # Re-define the global "globv"
+   globv <<- 2
+   cat("local globv =", globv, "\n")
+ }  # end side_effect
> side_effect()
> # Global variable was modified as side effect
> globv
```

Operators as Functions

Most functions in R are *prefix* operators (where the function name is followed by a list of arguments).

Infix operators (where the the function name comes in between its arguments) can also be applied using *prefix* syntax.

In *prefix* syntax, the *Infix* operator name must be surrounded by single '' or double "" quotes.

The "[" bracket operator can also be written as a *prefix* function.

```
> # Standard infix operator call syntax
> 2 + 3
> # Infix operator applied using prefix syntax
> "+"(2, 3)
> # Standard bracket operator
> vectorv <- c(4, 3, 5, 6)
> vectorv[2]
> # Bracket operator applied using prefix syntax
> "["(vectorv, 2)
>
```

Defining New Infix Operators

New *infix* operators can be defined using the usual function definition syntax.

All user defined *infix* operators names must be nested between "%" characters.

```
> # Define infix operator that returns string
> '%+%' <- function(a, b) paste(a, b, sep=" + ")
> 2 %+% 3
> 2 %+% 3 %+% 4
> "hello" %+% 2 %+% 3 %+% "bye"
```

Replacement Functions

R syntax allows assigning to the values returned by functions, but they must be defined as *replacement* functions.

replacement function names include the assignment arrow: "name<-".

The first argument passed to the *replacement* function is modified by the second argument, and then it's returned.

```
> obj_string <- "hello"
> class(obj_string)
> # Assign to value returned by "class" function
> class(obj_string) <- "string"
> class(obj_string)
> # Define function last()
> last <- function(vectorv) {
+   vectorv[NROW(vectorv)]
+ }  # end last
> last(1:10)
> # Define replacement function last()
> 'last<-' <- function(vectorv, value) {
+   vectorv[NROW(vectorv)] <- value
+   vectorv
+ }  # end last
> x <- 1:5
> last(x) <- 11
> x
```

Functions as First Class Objects

Functions in R are *first class objects*, which means they can be treated like any other R object:

- Functions can be passed as arguments to other functions,
- Functions can be nested (defined inside other functions),
- Functions can return functions as their return value,

Higher order functions are R functions that either accept a function as their argument (input) or return a function as their value (output).

```
> # Create functional that accepts a function as input argument
> testfun <- function(func_name) {
+ # Calculates statistic on random numbers
+   set.seed(1)
+   func_name(runif(1e4))  # Apply the function name
+ }  # end testfun
> testfun(mean)
> testfun(sd)
```

Functions That Return Functions

R functions can also return a function as their value.

Functions returned by a function are called *closures*.

Functions that return closures can be used as *function factories*.

```
> # Define a power function factory
> makefun <- function(arg_param) {  # Wrapper function
+   function(input) {  # Anonymous closure
+     input^arg_param
+   }
+ }  # end makefun
>
> squarefun <- makefun(2)  # Define square function
> squarefun(4)
> cubefun <- makefun(3)  # Define cube function
> cubefun(2)
> cube_rootfun <- makefun(1/3)  # Define cube root function
> cube_rootfun(8)
```

Mutable States

A *mutable state* is an object that is preserved between function calls.

Functions that return closures can also be used for creating *mutable states*.

A function *evaluation* environment is only temporary and disappears after the function returns its value.

But a *closure* assigned to a name maintains access to the environment in which it was created.

Therefore the *closure* maintains access to its parent function's arguments and locally defined variables.

```
> make_counter <- function() {
+ # Counter function with mutable state
+   counter <- 0  # Initialize counter
+   cat('counter = ', counter)
+   function() {  # Return anonymous advance function
+     counter <<- counter + 1  # Advance counter
+     cat('counter = ', counter)
+   }  # end advance function
+ }  # end make_counter
>
> advance_counter <- make_counter()  # Create new counter
> advance_counter()  # Advance counter
> advance_counter()  # Advance counter
> advance_counter_two <- make_counter()  # Create another counter
> advance_counter_two()  # Advance counter two
> advance_counter()  # Advance counter one
> advance_counter_two()  # Advance counter two
> advance_counter()  # Advance counter one
```

Pseudo-Random Generating Function

Mutable states can be used to implement pseudo-random number generators,

```
> # Returns the pseudo-random generating function random_generator
> # the formal argument 'seed' persists in the evaluation environment of seed_random
> seed_random <- function(seed) {  # Seed must be an integer
+   random_number <- as.numeric(paste0('0.', seed))  # Initialize
+ # Random_generator returns a vector of pseudo-random numbers of length length_rand
+   random_generator <- function(length_rand=1) {  # Assign function name for recursion
+ # Returns a vector of pseudo-random numbers of length length_rand
+     random_number <<- 4*random_number*(1 - random_number)  # Logistic map
+     if (length_rand == 1) {
+       return(random_number)
+     } else {
+       return(c(random_number, random_generator(length_rand - 1)))
+     }  # end if
+   }  # end random_generator
+ }  # end seed_random
>
> # Create a random number generating function and set seed
> make_random <- seed_random(88)
> make_random(10)  #  calculate vector of 10 pseudo-random numbers
> ls(environment(make_random))  # List objects in scope of make_random
```

Bank Account Using Mutable States

```
> # Bank account example (from Venables) demonstrates mutable state
> # 'balance' is persistent between function calls
> open_account <- function(balance) {
+ # Returns function list for account operations
+   list(
+     deposit = function(amount) {  # Make deposit
+       if (amount > 0) {
+ balance <<- balance + amount  # '<<-' super-assignment operator
+ cat(amount, "deposited. Your balance is now:",
+     balance, "\n")
+       } else {
+ cat("Deposits must be positive!\n")
+       }
+     },  # end deposit
+     withdraw = function(amount) {  # Make withdrawal
+       if (amount <= balance) {
+ balance <<- balance - amount  # '<<-' super-assignment operator
+ cat(amount, "withdrawn. Your balance is now:",
+     balance, "\n")
+       } else {
+ cat("You don't have that much money!\n")
+       }
+     },  # end withdraw
+     get_balance = function() {  # Get balance
+       cat("Your current balance is:", balance, "\n")
+     }  # end get_balance
+   )  # end list
+ }  # end open_account
```

```
> # Perform account operations
> # open an account with 100 deposit
> my_account <- open_account(100)
> ls(my_account)  # my_account is a list
> # Add my_account to search path
> attach(my_account)
> withdraw(30)  # Withdrawal to buy groceries
> deposit(100)  # Deposit paycheck to account
> withdraw(200)  # Withdrawal to buy Gucci bag
> get_balance()  # Get account balance
>
> # List objects in scope of get_balance
> ls(environment(get_balance))
>
> detach(my_account)  # Remove my_account from search path
```

Functionals

Functionals are functions that accept a function or a function name (string) as one of their input arguments.

Functionals are able to execute function calls using the function names.

The function `match.fun()` returns a function name that is specified by a string.

Functionals that call `match.fun()` are able to accept a string as a function name, because `match.fun()` converts it to a function.

`match.fun()` produces an error condition if it fails to find a function with the specified name.

```
> # Functional accepts function name and additional argument
> testfun <- function(func_name, input) {
+ # Produce function name from argument
+   func_name <- match.fun(func_name)
+ # Execute function call
+   func_name(input)
+ }  # end testfun
> testfun(sqrt, 4)
> # String also works because match.fun() converts it to a function
> testfun("sqrt", 4)
> str(sum)  # Sum() accepts multiple arguments
> # Functional can't accept indefinite number of arguments
> testfun(sum, 1, 2, 3)
```

Functionals with dots "..." Argument

The dots "..." argument in *functionals* can be used to pass additional arguments to the function being called by the *functional*.

If named values are passed to the dots "..." argument, then the *functional* can bind them to the correct formal arguments of the function being called by the *functional*.

```
> # Functional accepts function name and dots '...' argument
> testfun <- function(func_name, ...) {
+   func_name <- match.fun(func_name)
+   func_name(...)  # Execute function call
+ }  # end testfun
> testfun(sum, 1, 2, 3)
> testfun(sum, 1, 2, NA, 4, 5)
> testfun(sum, 1, 2, NA, 4, 5, na.rm=TRUE)
> # Function with three arguments and dots '...' arguments
> testfun <- function(input, param1, param2, ...) {
+   c(input=input, param1=param1, param2=param2, dots=c(...))
+ }  # end testfun
> testfun(1, 2, 3, param2=4, param1=5)
> testfun(testfun, 1, 2, 3, param2=4, param1=5)
> testfun(testfun, 1, 2, 3, 4, 5)
```

Anonymous Functions

R allows defining functions without assigning a name to them.

Anonymous functions are functions that are not assigned to a name.

Anonymous functions can be passed as arguments to *functionals*.

```
> # Simple anonymous function
> (function(x) (x + 3)) (10)
```

Functionals with Anonymous Functions

Anonymous functions can be passed as arguments to *functionals*.

Anonymous functions can also be used as default values for function arguments.

```
> # Anonymous function passed to testfun
> testfun(func_name=(function(x) (x + 3)), 5)
> # Anonymous function is default value
> testfun <-
+   function(..., func_name=function(x, y, z) {x+y+z}) {
+     func_name <- match.fun(func_name)
+     func_name(...)  # Execute function call
+ }  # end testfun
> testfun(2, 3, 4)  # Use default func_name
> testfun(2, 3, 4, 5)
> # Func_name bound by name
> testfun(func_name=sum, 2, 3, 4, 5)
> # Pass anonymous function to func_name
> testfun(func_name=function(x, y, z) {x*y*z},
+     2, 3, 4)
```

Executing Function Calls Using the do.call() Functional

The functional `do.call()` executes a function call using a function name and a list of arguments.

`do.call()` allows calling a function on arguments that are elements of a list.

`do.call()` passes the list elements individually, instead of passing the whole list as one argument: `do.call(fun, list)= fun(list[[1]], list[[2]], ...)`

`do.call()` can be called inside other *functionals* to allow them to execute function calls.

The function `str()` displays the structure of an `R` object, for example a function name and its formal arguments.

The function `do_call()` from package *rutils* performs the same operation as `do.call()`, but using recursion, which is much faster and uses less memory.

```
> str(sum)  # Sum() accepts multiple arguments
> # Sum() can't accept list of arguments
> sum(list(1, 2, 3))
> str(do.call)  # "what" argument is a function
> # Do.call passes list elements into "sum" individually
> do.call(sum, list(1, 2, 3))
> do.call(sum, list(1, 2, NA, 3))
> do.call(sum, list(1, 2, NA, 3, na.rm=TRUE))
> # Functional accepts list with function name and arguments
> testfun <- function(list_arg) {
+ # Produce function name from argument
+   func_name <- match.fun(list_arg[[1]])
+ # Execute function call uing do.call()
+   do.call(func_name, list_arg[-1])
+ }  # end testfun
> arg_list <- list("sum", 1, 2, 3)
> testfun(arg_list)
> # do_call() performs same operation as do.call()
> all.equal(
+   do.call(sum, list(1, 2, NA, 3, na.rm=TRUE)),
+   rutils::do_call(sum, list(1, 2, NA, 3), na.rm=TRUE))
```

Performing Loops Using the apply() Functionals

An important example of *functionals* are the apply() functionals.

The functional apply() returns the result of applying a function to the rows or columns of an array or matrix.

If MARGIN=1 then the function will be applied over the matrix *rows*,

If MARGIN=2 then the function will be applied over the matrix *columns*.

apply() performs a loop over the list of objects, and can replace "for" loops in R.

```
> str(apply)  # Get list of arguments
> # Create a matrix
> matrixv <- matrix(6:1, nrow=2, ncol=3)
> matrixv
> # Sum the rows and columns
> rowsumv <- apply(matrixv, 1, sum)
> colsumv <- apply(matrixv, 2, sum)
> matrixv <- cbind(c(sum(rowsumv), rowsumv),
+           rbind(colsumv, matrixv))
> dimnames(matrixv) <- list(c("colsumv", "row1", "row2"),
+                   c("rowsumv", "col1", "col2", "col3"))
> matrixv
```

The apply() Functional with dots "..." Argument

The dots "..." argument in apply() is designed to pass additional arguments to the function being called by apply().

The additional arguments to apply() must be *bound* by their full (complete) names.

```
> str(apply)  # Get list of arguments
> matrixv <- matrix(sample(12), nrow=3, ncol=4)  # Create a matrix
> matrixv
> apply(matrixv, 2, sort)  # Sort matrix columns
> apply(matrixv, 2, sort, decreasing=TRUE)  # Sort decreasing order
```

```
> matrixv[2, 2] <- NA  # Introduce NA value
> matrixv
> # Calculate median of columns
> apply(matrixv, 2, median)
> # Calculate median of columns with na.rm=TRUE
> apply(matrixv, 2, median, na.rm=TRUE)
```

The apply() Functional with Anonymous Functions

The apply() functional combined with *anonymous* functions can be used to loop over function parameters.

The dots "..." argument in apply() is designed to pass additional arguments to the function being called by apply().

The additional arguments to apply() must be *bound* by their full (complete) names.

```
> # VTI percentage returns
> retp <- rutils::diffit(log(Cl(rutils::etfenv$VTI)))
> library(moments)  # Load package moments
> str(moment)  # Get list of arguments
> # Apply moment function
> moment(x=retp, order=3)
> # 4x1 matrix of moment orders
> orderv <- as.matrix(1:4)
> # Anonymous function allows looping over function parameters
> apply(X=orderv, MARGIN=1,
+ FUN=function(orderp) {
+     moment(x=retp, order=orderp)
+   }  # end anonymous function
+ )  # end apply
>
> # Another way of passing parameters into moment() function
> apply(X=orderv, MARGIN=1, FUN=moment, x=retp)
```

apply() Calling Functions with Multiple Arguments

When apply() calls a function with multiple arguments, then care must be taken for proper argument binding.

The dots "..." argument in apply() allows passing additional arguments to the function being called by apply().

The additional arguments to apply() must be *bound* by their full (complete) names.

The values of the "X" argument in apply() are *bound* by position to the first unused argument in the function being called by apply().

```
> # Function with three arguments
> testfun <- function(arg1, arg2, arg3) {
+   c(arg1=arg1, arg2=arg2, arg3=arg3)
+ }  # end testfun
> testfun(1, 2, 3)
> datav <- as.matrix(1:4)
> # Pass datav to arg1
> apply(X=datav, MAR=1, FUN=testfun, arg2=2, arg3=3)
> # Pass datav to arg2
> apply(X=datav, MAR=1, FUN=testfun, arg1=1, arg3=3)
> # Pass datav to arg3
> apply(X=datav, MAR=1, FUN=testfun, arg1=1, arg2=2)
```

The lapply() Functional

The functional `lapply()` is a specialized version of the functional `apply()`.

`lapply()` applies a function to a list of objects and returns a list.

The function `unlist()` collapses a list with atomic elements into a vector (which can cause type coercion).

Rule of Thumb

It's often better to use `lapply()`, since `apply()` and `sapply()` attempt to coerce their output into a vector or matrix, which may cause them to fail.

```
> # Vector of means of numeric columns
> sapply(iris[, -5], mean)
> # List of means of numeric columns
> lapply(iris[, -5], mean)
> # Lapply using anonymous function
> unlist(lapply(iris,
+        function(column) {
+          if (is.numeric(column)) mean(column)
+        }  # end anonymous function
+        )  # end lapply
+         )  # end unlist
> unlist(sapply(iris, function(column) {
+   if (is.numeric(column)) mean(column)}))
```

The sapply() Functional

The sapply() functional is a specialized version of the apply() functional.

sapply() applies a function to a vector or a list of objects and returns a vector or a list.

sapply() tries to return a vector, but if the elements can't be combined into a vector, then it returns a list.

When sapply() is given a data frame, it interretsp it as a list, and applies the function to each element (column) of the data frame.

```
> sapply(6:10, sqrt)  # Sapply on vector
> sapply(list(6, 7, 8, 9, 10), sqrt)  # Sapply on list
>
> # Calculate means of iris data frame columns
> sapply(iris, mean)  # Returns NA for Species
>
> # Create a matrix
> matrixv <- matrix(sample(100), ncol=4)
> # Calculate column means using apply
> apply(matrixv, 2, mean)
>
> # Calculate column means using sapply, with anonymous function
> sapply(1:NCOL(matrixv), function(colnum) {  # Anonymous function
+  mean(matrixv[, colnum])
+   }  # end anonymous function
+ )  # end sapply
```

sapply() Returning Matrices

If the function called by sapply() returns a vector, then sapply() returns a matrix, if possible.

The vectors returned by the function are arranged to form columns of the matrix returned by sapply().

But if the function returns vectors of different lengths, then sapply() cannot return a matrix, and returns a list instead.

This behavior of sapply() can cause run-time errors.

The function vapply() is similar to sapply(), but it always attempts to simplify its output to a matrix, and if it can't then it produces an error.

vapply() requires the argument FUN.VALUE that specifes the output format of the function called by vapply().

```
> # Vectors form columns of matrix returned by sapply
> sapply(2:4, function(num) c(el1=num, el2=2*num))
> # Vectors of different lengths returned as list
> sapply(2:4, function(num) 1:num)
> # vapply is similar to sapply
> vapply(2:4, function(num) c(el1=num, el2=2*num),
+        FUN.VALUE=c(row1=0, row2=0))
> # vapply produces an error if it can't simplify
> vapply(2:4, function(num) 1:num,
+        FUN.VALUE=c(row1=0, row2=0))
```

The S3 Object-Oriented Programming System in R

S3 is the standard object oriented (OO) programming system in R.

The S3 system is based on *generic* functions and the R *class* system.

Generic functions are functions that execute different *methods* depending on the class of the object on which the *generic* function is called.

Methods are functions that are specific to a *generic* function and a class of objects.

Methods follow the naming convention `generic_function.classname()`.

The actual function that is executed (called a *method*) is determined by the class of the object on which the *generic* function is called.

For example, when the function `merge()` is called on a `zoo` object, then R executes the *method* `merge.zoo()`.

```
> library(zoo)  # Load package zoo
> # Show the generic function "merge"
> merge
> # Show the "merge" method dispatched to "zoo" objects
> merge.zoo
```

Generic Functions and Their Methods

The *generic* function `merge()` has many *methods* with names `merge.*()`.

The function `methods()` listv all the *methods* of a generic function, or all the *methods* for a *class* of objects.

The `merge()` method dispatched to zoo objects is called `merge.zoo()`.

```
> # Get all methods for generic function merge()
> methods(generic.function="merge")
> # Get generic function methods applied to "zoo" objects
> methods(class="zoo")
```

Method Dispatch Using UseMethod()

The function `UseMethod()` can be used to implement *generic* functions.

`UseMethod()` accepts at least two arguments: the name of a *generic* function, and the arguments passed to the *generic* function.

`UseMethod()` calls (*dispatches*) a particular *method* associated with the *generic* function, depending on the *class* of the arguments passed to the *generic* function.

The arguments passed to the *generic* function are by default passed to `UseMethod()`, and then along to the *method* itself.

```
> # Define a generic function
> gen_sum <- function(a, b, ...) {
+   UseMethod("gen_sum")
+ }  # end gen_sum
>
> # Define method for "numeric" class
> gen_sum.numeric <- function(a, b, ...) {
+   sum(a, b)
+ }  # end gen_sum.character
>
> # Define method for "character" class
> gen_sum.character <- function(a, b, ...) {
+   paste(a, "plus", b)
+ }  # end gen_sum.character
>
> # Apply gen_sum to "numeric" objects
> gen_sum(1, 2)
> # Apply gen_sum to "character" objects
> gen_sum("a", "b")
```

Method Dispatch by Internal Generic Functions

Method dispatch by *internal generic* functions is performed inside compiled `C` code, instead of `R` code using the function `UseMethod()`.

Internal functions are implemented using the function `.Internal()`.

```
> # 'cbind' is an internal generic function
> cbind
```

Operator Overloading

Operator *overloading* refers to defining new *methods* for an existing *generic* function.

The "+" operator may be overloaded by defining a new *method* for "character" objects.

But for the *overloading* of the "+" operator to work, the objects must have an explicit "character" class *attribute* assigned to them.

```
> # Define "+" method for "character" class
> "+.character" <- function(a, b, ...) {
+   paste(a, "plus", b)
+ }  # end +.character
> methods("+")  # view methods for "+" operator
> # Define variables with "character" class
> char1 <- "a"
> char2 <- "b"
> class(char1)
> char1 + char2  # Add two "character" objects - doesn't work
> attributes(char1)  # Doesn't have explicit "character" class - onl
> char1 <- structure("a", class="character")
> char2 <- structure("b", class="character")
> attributes(char1)  # Now has explicit "character" class
> # Add two "character" objects
> char1 + char2
```

Overloading the print() Function

The *generic* functions `print()`, `plot()` and `summary()` are very often *overloaded* for newly defined classes.

Since `print()` is a *generic* function, R *dispatches* the *method* associated with the *class* of that variable.

When a variable is called by its name, then R invokes the `print()` function on that variable.

```
> # Define object of class "string"
> obj_string <- "how are you today?"
> class(obj_string) <- "string"
> obj_string
> # overload "print" method for string objects
> print.string <- function(str_ing) {
+   print(
+     paste(strsplit(str_ing, split=" ")[[1]],
+   collapse=" + "))
+ }  # end print.string
> # methods("print")  # view new methods for "print" function
> print(obj_string)
> obj_string
```

Operator Overwriting

Operator *overwriting* refers to redefining an existing function.

The functions `.Internal()` and `.Primitive()` call functions that are part of the internal code of R.

Operator *overwriting* should be used with care, since it may cause unintended consequences.

```
> # overwrite "+" operator
> "+" = function(a, b) {
+   if (is.character(a) && is.character(b)) {
+     paste(a, "plus", b)
+   } else {
+     .Primitive("+") (a, b)
+   }
+ }
> methods("+")  # view methods for "+" operator
> # Add two "numeric" objects
> 1 + 2
> # Add two "character" objects
> "a" + "b"
```

Operator Overwriting Using UseMethod()

Existing functions can be *overwritten* with *generic* functions using `UseMethod()`.

Operator *overwriting* should be used with care, since it may cause unintended consequences.

```
> # overwrite "+" operator with a generic function
> "+" <- function(a, b, ...) {
+   UseMethod("+")
+ }  # end gen_sum
> # Define method for "numeric" class
> "+.numeric" <- function(a, b, ...) {
+   sum(a, b)
+ }  # end gen_sum.character
> # Define method for "character" class
> "+.character" <- function(a, b, ...) {
+   paste(a, "plus", b)
+ }  # end gen_sum.character
> methods("+")  # view methods for "+" operator
> # Add two "numeric" objects
> 1 + 2
> # Add two "character" objects
> "a" + "b"
```

Exploring Generic Function Methods

Most *methods* can be viewed by simply calling their full name, unless they're non-visible.

Non-visible *methods* can be viewed using the triple-colon operator `":::"`.

Non-visible *methods* can also be viewed by calling the function `getAnywhere()`.

```
> cbind.ts  # Can't view non-visible method
> stats::cbind.ts  # Can't view non-visible method
> stats:::cbind.ts  # Display non-visible method
> getAnywhere(cbind.ts)  # Display non-visible method
```

Defining New Classes and Methods

A new R class can be created by simply assigning to the *class* attribute of an existing object.

New *methods* can be defined for existing *generic* functions, and R will automatically *dispatch* them for objects of the new *class*.

The function unclass() removes the explicit class attribute from an object.

Calling unclass() allows using the *methods* associated with the original object before a new *class* attribute was assigned to it.

The functions `.Internal()` and `.Primitive()` call internally implemented (*primitive*) functions.

```
> new_zoo <- zoo(rnorm(10), order.by=(Sys.Date() + 0:9))
> # Coerce "zoo" object to new class "zoo_xtra"
> class(new_zoo) <- "zoo_xtra"
> class(new_zoo)
> methods(generic.function="length")
> length  # Primitive function
> # Define "length" method for class "zoo_xtra"
> length.zoo_xtra <- function(in_ts) {
+   cat("length of zoo_xtra object:\n")
+ # Unclass object, then calculate length
+   NROW(unclass(in_ts))
+ }  # end length.zoo_xtra
> NROW(new_zoo)  # Apply "length" method to "zoo_xtra" object
> methods(generic.function="length")
```

Defining New Generic Functions and Methods

New *methods* have to be called by their full name if a *generic* function isn't defined for them.

Once a *generic* function is defined, then new *methods* can be called by their short name

```
> # Define "last" method for class "zoo_xtra"
> last.zoo_xtra <- function(in_ts) {
+   in_ts[NROW(in_ts)]
+ }  # end last.zoo_xtra
> last(new_zoo)  # Doesn't work
> last.zoo_xtra(new_zoo)  # Works
> # Define a generic function
> last <- function(a, b, ...) {
+   UseMethod("last")
+ }  # end last
> last(new_zoo)  # Now works
```

Creating a "string" Class

A new `"string"` class can be created from a `character` object, by assigning to its *class* attribute.

The *generic* function `as.string()` converts objects to class "*string*".

The function `structure()` adds attributes to an object (specified as `symbol=value` pairs), and returns it.

The function `inherits()` checks whether the object *class* matches any of the names in the `"what"` argument.

```
> # Define generic "string" class converter
> as.string <- function(str_ing, ...)
+   UseMethod("as.string")
> # Default "string" class converter
> as.string.default <- function(str_ing, ...)
+   structure(str_ing, class="string", ...)
> # Numeric "string" class converter
> as.string.numeric <- function(str_ing, ...)
+   structure(as.character(str_ing), class="string", ...)
> # "string" class checker
> is.string <- function(str_ing)
+   inherits(x=str_ing, what="string")
> # Define "string" object
> obj_string <- as.string("how are you today?")
> obj_string
> is.string(obj_string)
> is.string("hello")
> as.string(123)
> is.string(as.string(123))
```

Inheritance and Derived Classes and Methods

Inheritance is a mechanism for defining a new class that is *derived* from a *base* class.

The *derived* class *inherits* all the *methods* from the *base* class, but can also have new *methods* of its own.

In the S3 system *inheritance* is implemented by making the *class* attribute a *vector*.

When a *generic* function `gen_fun` is called on an object with class attribute `c("class2", "class1")`, then R *dispatches* a *method* called `gen_fun.class2`.

If there's no *method* with that name, then R first *dispatches* a *method* called `gen_fun.class1`.

Finally if there are no *methods* with those names, then R *dispatches* a *method* called `gen_fun.default`.

```
> library(xts)
> new_xts <- xts(rnorm(10), order.by=(Sys.Date() + 0:9))
> class(new_xts)  # Class attribute is a vector
> # "last" is a generic function from package "xts"
> last
> methods(generic.function="last")
> last(new_xts)  # Apply "last" method from "xts" class
> # Derive object "xts_xtra" from "xts" object
> class(new_xts) <- c("xts_xtra", class(new_xts))
> class(new_xts)  # Class attribute is a vector
> # "xts_xtra" object inherits "last" method from "xts" class
> last(new_xts)
```

Defining New Methods for Derived Classes

The S3 system automatically *dispatches* newly defined *methods* to objects of the new *class*.

If new *methods* aren't found, then it *dispatches* existing *methods* from the *base* class to objects of the new *class*.

The function `NextMethod()` *dispatches* the base method of a *generic* function.

```
> # Define new "last" method for class "xts_xtra"
> last.xts_xtra <- function(in_ts) {
+   cat("last element of xts_xtra object:\n")
+   drop(in_ts[NROW(in_ts), ])
+ }  # end last.xts_xtra
> last(new_xts)  # Apply "last" from "xts_xtra" class
> # Define "last" method for class "xts_xtra"
> last.xts_xtra <- function(in_ts) {
+   cat("last element of xts_xtra object:\n")
+   drop(NextMethod())
+ }  # end last.xts_xtra
> last(new_xts)  # Apply "last" from "xts_xtra" class
```

Homework Assignment

Required

- Create a function for calculating the kurtosis of a time series of returns,
- Using this function calculate the kurtosis of DAX returns, and of t-distribution returns with four degrees of freedom (use the same number of data points in both cases),
- Plot the probability density of DAX returns together with t-distribution returns with four degrees of freedom on a single plot,

Homework Assignment

Required

- Create a function for calculating the kurtosis of a time series of returns,
- Using this function calculate the kurtosis of DAX returns, and of t-distribution returns with four degrees of freedom (use the same number of data points in both cases),
- Plot the probability density of DAX returns together with t-distribution returns with four degrees of freedom on a single plot,

Recommended

- Read chapters 4, 5, 10 from: *Introduction to R*.

Additional Reading

Download R Interpreter from CRAN (Comprehensive R Archive Network)

http://cran.r-project.org/

Download *RStudio* IDE (Integrated Development Environment)

http://www.rstudio.com/products/rstudio/