R Programming Cheat Sheet

ADVANCED

Created By: Arianne Colton and Sean Chen

Environments

ENVIRONMENT BASICS

What is an Environment?

Data structure (that powers lexical scoping) is made up of two components, the frame, which contains the name-object bindings (and behaves much like a named list), and the parent environment.

Named List

- You can think of an environment as a bag of names. Each name points to an object stored elsewhere in memory.
- If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

Parent Environment

- Every environment has a parent, another environment. Only one environment doesn't have a parent: the empty environment.
- The parent is used to implement lexical scoping: if a name is not found in an environment, then R will look in its parent (and so on).

Environments can also be useful data structures in their own right because they have reference semantics.

FOUR SPECIAL ENVIRONMENTS

1. **Global environment**, access with `globalenv()`, is the interactive workspace. This is the environment in which you normally work.

   The parent of the global environment is the last package that you attached with `library()` or `require()`.

2. **Base environment**, access with `baseenv()`, is the environment of the base package. Its parent is the empty environment.

3. **Empty environment**, access with `emptyenv()`, is the ultimate ancestor of all environments, and the only environment without a parent. Empty environments contain nothing.

4. **Current environment**, access with `environment()`

SEARCH PATH

What is the Search Path?

An R internal mechansim to look up objects, specifically, functions.

- Access with `search()`, which lists all parents of the global environment. (See Figure 1)
- It contains one environment for each attached package.
- Objects in the search path environments can be found from the top-level interactive workspace.

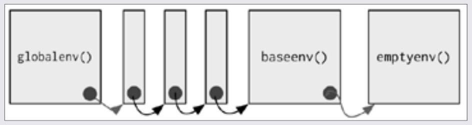

Figure 1. *The Search Path*

- If you look for a name in a search, it will always start from global environment first, then inside the latest attached package.

  If there are functions with the same name in two different packages, the latest package will get called.

- Each time you load a new package with `library()`/ `require()` it is inserted between the global environment and the package that was previously at the top of the search path.

```
search() :
'.GlobalEnv' ... 'Autoloads' 'package:base'
library(reshape2); search()
'.GlobalEnv'  'package:reshape2' ...
'Autoloads' 'package:base'
```

Note: There is a special environment called Autoloads which is used to save memory by only loading package objects (like big datasets) when needed.

Environments

Access any environment on the search list	`as.environment('package:base')`
Find the environment where a name is defined	`pryr::where('func1')`

FUNCTION ENVIRONMENTS

There are 4 environments for functions.

1. **Enclosing environment (used for lexical scoping)**
   - When a function is created, it gains a reference to the environment where it was made. This is the enclosing environment.
   - The enclosing environment belongs to the function, and never changes, even if the function is moved to a different environment.
   - Every function has one and only one enclosing environment. For the three other types of environment, there may be 0, 1, or many environments associated with each function.
   - You can determine the enclosing environment of a function by calling i.e. `environment(func1)`

2. **Binding environment**
   - The binding environments of a function are all the environments which have a binding to it.
   - The enclosing environment determines how the function finds values; the binding environments determine how we find the function.

Example for enclosing and binding environment

```
y <- 1
e <- new.env()
e$g <- function(x) x + y
```

# function g enclosing environment is the global environment, and the binding environment is "e".

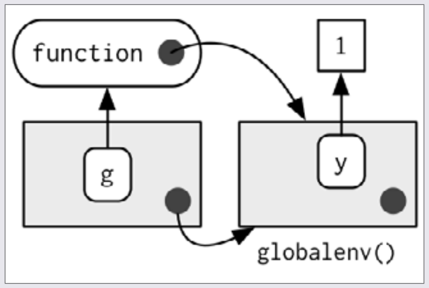

Figure 2. *Function Environment*

Note: Every R package has two environments associated with it (package and namespace). Every exported function is bound into the package environment, but enclosed by the namespace environment.

3. **Execution environment**
   - Each time a function is called, a new environment is created to host execution. The parent of the execution environment is the enclosing environment of the function.
   - Once the function has completed, this environment is thrown away.

   Note: Each execution environment has two parents: a calling environment and an enclosing environment.

   - R's regular scoping rules only use the enclosing parent; parent.frame() allows you to access the calling parent.

4. **Calling environment**
   - This is the environment where the function was called.
   - Looking up variables in the calling environment rather than in the enclosing environment is called dynamic scoping.
   - Dynamic scoping is primarily useful for developing functions that aid interactive data analysis.

BINDING NAMES TO VALUES

Assignment

- Assignment is the act of binding (or rebinding) a name to a value in an environment.

Name rules

- A complete list of reserved words can be found in `?Reserved`.

Regular assignment arrow, <-

- The regular assignment arrow always creates a variable in the current environment.

Deep assignment arrow, <<-

- The deep assignment arrow modifies an existing variable found by walking up the parent environments. If <<- doesn't find an existing variable, it will create one in the global environment. This is usually undesirable, because global variables introduce non-obvious dependencies between functions.

ENVIRONMENT CREATION

- To create an environment manually, use `new.env()`. You can list the bindings in the environment's frame with `ls()` and see its parent with `parent.env()`.
- When creating your own environment, note that you should set its parent environment to be the empty environment. This ensures you don't accidentally inherit objects from somewhere else.

Functions

FUNCTION BASICS

The most important thing to understand about R is that **functions are objects** in their own right.

All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	"map" of the location of the function's variables (see "Enclosing Environment")

- When you `print(func1)` a function in R, it shows you these three important components. If the environment isn't displayed, it means that the function was created in the global environment.
- Like all objects in R, functions can also possess any number of additional attributes().

Every operation is a function call

- Everything that exists is an object
- Everything that happens in R is a function call, even if it doesn't look like it. (i.e. +, for, if, [, $, { ...)

Note: the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names: e.g. `x + y` is the same as `` `+`(x, y) ``

LEXICAL SCOPING

What is Lexical Scoping?

Looks up value of a symbol. (See "Enclosing Environment" in the "Environment" section.)

- `findGlobals()` # lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function "+" *
```

* This doesn't work because R relies on lexical scoping to find everything, even the + operator. It's never possible to make a function completely self-contained because you must always rely on functions defined in base R or other packages.

FUNCTION ARGUMENTS

When calling a function you can specify arguments by position, by complete name, or by partial name. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

- Function arguments are passed by **reference** and **copied on modify**.
- You can determine if an argument was supplied or not with the missing() function.

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

- By default, R function arguments are lazy -- they're only evaluated if they're actually used

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

However, since x is not used. `stop("This is an error!")` never get evaluated.

- Default arguments are evaluated inside the function. This means that if the expression depends on the current environment the results will differ depending on whether you use the default value or explicitly provide one:

```
f <- function(x = ls()) {
   a <- 1
   x
}
```

`f() -> 'a' 'x'`	ls() evaluated inside f
`f(ls())`	ls() evaluated in global environment

RETURN VALUES

- The last expression evaluated in a function becomes the return value, the result of invoking the function.
- Only use explicit `return()` for when you are returning early, such as for an error.
- Functions can return only a single object. But this is not a limitation because you can return a list containing any number of objects.
- Functions can return invisible values, which are not printed out by default when you call the function.

```
f1 <- function() 1
f2 <- function() invisible(1)
```

- The most common function that returns invisibly is <-

PRIMITIVE FUNCTIONS

- There is one exception to the rule that functions have three components.
- Primitive functions, like `sum()`, call C code directly with `.Primitive()` and contain no R code.
- Therefore their formals(), body(), and environment() are all NULL:

```
sum : function (..., na.rm = FALSE)
.Primitive('sum')
```

- Primitive functions are only found in the base package, and since they operate at a low level, they can be more efficient.

INFIX FUNCTIONS

- Most functions in R are 'prefix' operators: the name of the function comes before the arguments.
- You can also create infix functions where the function name comes in between its arguments, like + or -.
- All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

- Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.
null(a)) a else b
function_that_might_return_null() %||%
default value
```

REPLACEMENT FUNCTIONS

- Act like they modify their arguments in place, and have the special name `xxx <-`
- They typically have two arguments (x and value), although they can have more, and they must return the modified object.

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```

- I say they "act" like they modify their arguments in place, because they actually create a modified copy.
- We can see that by using `pryr::address()` to find the memory address of the underlying object.

Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Note: R has no 0-dimensional or scalar types. Individual numbers or strings, are actually vectors of length one, NOT scalars.

Human readable description of any R data structure:

```
str(variable)
```

Every **Object** has a mode and a class

1. **Mode**: represents how an object is stored in memory;
   - 'type' of the object from R's point of view
   - Access with `typeof()`
2. **Class**: represents the object's abstract type;
   - 'type' of the object from R's object-oriented programming point of view
   - Access with `class()`

	typeof()	class()
strings or vector of strings	character	character
numbers or vector of numbers	numeric	numeric
list	list	list
data.frame*	list	data.frame

* Internally, data.frame is a list of equal-length vectors.

1D (VECTORS: ATOMIC VECTOR AND LIST)

- Use `is.atomic() || is.list()` to test if an object is actually a vector, not `is.vector()`.

Type	typeof()	what it is
Length	length()	how many elements
Attributes	attributes()	additonal arbitrary metadata

FACTORS

- Factors are built on top of integer vectors using two attributes :

`class(x) -> 'factor'`
`levels(x)` # defines the set of allowed values

- While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings.
- Factors are useful when you know the possible values a variable may take, even if you don't see all values in a given dataset.
- Most data loading functions in R automatically convert character vectors to factors, use the argument `stringsAsFactors = FALSE` to suppress this behavior.

ATTRIBUTES

- All objects can have arbitrary additional attributes.
- Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

```
attr(v1, 'attr1') <- 'my vector'
```

- By default, most attributes are lost when modifying a vector. The only attributes not lost are the three most important:

Names	a character vector giving each element a name	`names(x)`
Dimensions	used to turn vectors into matrices and arrays	`dim(x)`
Class	used to implement the S3 object system	`class(x)`

Subsetting (operators: [, [[, $)

Simplifying vs. preserving subsetting

- **Simplifying** subsetting returns the **simplest** possible data structure that can represent the output.
- **Preserving** subsetting keeps the structure of the output the **same** as the input.

	Simplifying*	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = T]`	`x[1:4]`
Array	`x[1, ]` or `x[, 1]`	`x[1, , drop = F]` or `x[, 1, drop = F]`
Data frame	`x[, 1]` or `x[[1]]`	`x[, 1, drop = F]` or `x[1]`

- When you use `drop = FALSE`, it's preserving.
- Omitting `drop = FALSE` when subsetting matrices and data frames is one of the most common sources of programming errors.
- `[[` is similar to `[`, except it can only return a single value and it allows you to pull pieces out of a list.

* Simplifying behavior varies slightly between different data types:

- **Atomic Vector**: `x[[1]]` is the same as `x[1]`.
- **List**: [ ] always returns a list, to get the contents use [[ ]].
- **Factor**: drops any unused levels but it remains a factor class.
- **Matrix or array**: if any of the dimensions has length 1, drops that dimension.
- **Data.frame** is similar, if output is a single column, it returns a vector instead of a data frame.

Data.frames subsetting

- Data frames possess the **characteristics of both lists and matrices**. If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices.

List Subsetting	`df1[c('col1', 'col2')]`
Matrix Subsetting	`df1[, c('col1', 'col2')]`

The subsetting results are the **same** in this example.

- **Single column subsetting**: matrix subsetting simplifies by default, list subsetting does not.

```
str(df1[, 'col1']) -> int [1:3]
# the result is a vector
```

```
str(df1['col1']) -> 'data.frame'
# the result remains a data frame of 1 column
```

Subsetting returns a copy of the original data, NOT copy-on-modified.

Out of bounds

- [and [[differ slightly in their behavior when the index is out of bounds (OOB).
- For example, when you try to extract the fifth element of a length four vector, aka OOB `x[5] -> NA`, or subset a vector with NA or NULL: `x[NULL] -> x[0]`

Operator	Index	Atomic	List
[	OOB	NA	list(NULL)
[	NA_real_	NA	list(NULL)
[	NULL	x[0]	list(NULL)
[[	OOB	Error	Error
[[	NA_real_	Error	NULL
[[	NULL	Error	Error

- If the input vector is named, then the names of OOB, missing, or NULL components will be "<NA>".

$ Subsetting operator

- $ is a useful shorthand for `[[` combined with character subsetting:

`x$y` is equivalent to `x[['y', exact = FALSE]]`

- One common mistake with `$` is to try and use it when you have the name of a column stored in a variable:

```
var <- 'cyl'
x$var
# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

- There's one important difference between `$` and `[[`, `$` does partial matching, `[[` does not:

```
x <- list(abc = 1)
x$a -> 1      # since "exact = FALSE"
x[['a']] -> # would be an error
```

Subsetting with assignment

- All subsetting operators can be combined with assignment to modify selected values of the input vector.
- Subsetting with nothing can be useful in conjunction with assignment because it will preserve the original object class and structure.

```
df1[] <- lapply(df1, as.integer)
# df1 will remain as a data frame
```

```
df1 <- lapply(df1, as.integer)
# df1 will become a list
```

Examples

1. **Lookup tables** (character subsetting)

   Character matching provides a powerful way to make lookup tables.

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)

lookup[x]
> m  f  u  f  f  m  m
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
unname(lookup[x])
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
```

2. **Matching and merging by hand** (integer subsetting)

   Lookup table which has multiple columns of information.

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
    grade = 3:1,
    desc = c('Excellent', 'Good', 'Poor'),
    fail = c(F, F, T)
)
```

First method :

```
id <- match(grades, info$grade)
info[id, ]
```

Second method :

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

- If you have multiple columns to match on, you'll need to first collapse them to a single column (with `interaction()`, `paste()`, or `plyr::id()`).
- You can also use `merge()` or `plyr::join()`, which do the same thing for you.

3. **Expanding aggregated counts** (integer subsetting)

- Sometimes you get a data frame where identical rows have been collapsed into one and a count column has been added.
- `rep()` and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: `rep(x, y)`
- rep replicates the values in x, y times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

4. **Removing columns from data frames** (character subsetting)

   There are two ways to remove columns from a data frame.

Set individual columns to NULL	`df1$col3 <- NULL`
Subset to return only the columns you want	`df1[c('col1', 'col2')]`

5. **Selecting rows based on a condition** (logical subsetting)

- Logical subsetting is probably the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$col1 == 5 & df1$col2 == 4, ]
```

- Remember to use the vector boolean operators `&` and `|`, not the short-circuiting scalar operators `&&` and `||` which are more useful inside if statements.
- `subset()` is a specialised shorthand function for subsetting data frames, and saves some typing because you don't need to repeat the name of the data frame.

```
subset(df1, col1 == 5 & col2 == 4)
```

Boolean algebra vs. sets (logical & integer subsetting)

- It's useful to be aware of the natural equivalence between set operations (integer subsetting) and boolean algebra (logical subsetting).
- Using set operations is more effective when:
  - » You want to find the first (or last) TRUE.
  - » You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage.
- `which()` allows you to convert a boolean representation to an integer representation. There's no reverse operation in base R.

```
which(c(T, F, T F)) -> 1 3
# returns the index of the true*
```

* The integer representation length is always <= boolean representation length.

- When first learning subsetting, a common mistake is to use `x[which(y)]` instead of `x[y]`.
- Here the `which()` achieves nothing, it switches from logical to integer subsetting but the result will be exactly the same.
- Also beware that `x[-which(y)]` is not equivalent to `x[!y]`. If y is all FALSE, `which(y)` will be `integer(0)` and `-integer(0)` is still `integer(0)`, so you'll get no values, instead of all values.
- In general, avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value.

Debugging, condition handling, & defensive programming

DEBUGGING

Use `traceback()` and `browser()`, and interactive tools in RStudio:

- RStudio's error inspector or `traceback()` which list the sequence of calls that lead to the error.
- RStudio's breakpoints or `browser()` which open an interactive debug session at an arbitrary location in the code.
- RStudio's "Rerun with Debug" tool or `options(error = browser)`* which open an interactive debug session where the error occurred.

* There are two other useful functions that you can use with the error option:

1. **Recover** is a step up from browser, as it allows you to enter the environment of any of the calls in the call stack.

   This is useful because often the root cause of the error is a number of calls back.

2. **dump.frames** is an equivalent to recover for non-interactive code. It creates a *last.dump.rda* file in the current working directory.

   Then, in a later interactive R session, you load that file, and use `debugger()` to enter an interactive debugger with the same interface as `recover()`. This allows interactive debugging of batch code.

   In batch R process ----

```
dump_and_quit <- function() {
   # Save debugging info to file last.dump.rda
   dump.frames(to.file = TRUE)

   # Quit R with error status
   q(status = 1)
}
options(error = dump_and_quit)
```

   In a later interactive session ----

```
load("last.dump.rda")
debugger()
```

CONDITION HANDLING (OF EXPECTED ERRORS)

1. **Communicating potential problems** to the user is the job of conditions: errors, warnings, and messages:

   - Fatal errors are raised by `stop()` and force all execution to terminate. Errors are used when there is no way for a function to continue.
   - Warnings are generated by `warning()` and are used to display potential problems, such as when some elements of a vectorised input are invalid.
   - Messages are generated by `message()` and are used to give informative output in a way that can easily be suppressed by the user using `?suppressMessages()`.

2. **Handling conditions programmatically**:

   - `try()` gives you the ability to continue execution even when an error occurs.
   - `tryCatch()` lets you specify handler functions that control what happens when a condition is signaled.

```
result = tryCatch(code,
     error = function(c) "error",
     warning = function(c) "warning",
     message = function(c) "message"
)
```

   Use `conditionMessage(c)` or `c$message` to extract the message associated with the original error.

   - You can also capture the output of the `try()` and `tryCatch()` functions.

     If successful, it will be the last result evaluated in the block, just like a function.

     If unsuccessful it will be an invisible object of class "try-error".

3. **Custom signal classes**:

   - One of the challenges of error handling in R is that most functions just call `stop()` with a string.
   - Since conditions are S3 classes, the solution is to define your own classes if you want to distinguish different types of error.
   - Each condition signalling function, `stop()`, `warning()`, and `message()`, can be given either a list of strings, or a custom S3 condition object.

DEFENSIVE PROGRAMMING

The basic principle of defensive programming is to "**fail fast**", to raise an error as soon as something goes wrong.

In R, this takes three particular forms:

1. Checking that inputs are correct using `stopifnot()`, the 'assertthat' package, or simple if statements and `stop()`
2. Avoiding non-standard evaluation like `subset()`, `transform()`, and `with()`.

   These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages.
3. Avoiding functions that can return different types of output. The two biggest offenders are `[` and `sapply()`.

Note: Whenever subsetting a data frame in a function, you should always use `drop = FALSE`

Object Oriented (OO) Field Guide

OBJECT ORIENTED SYSTEMS

R has three object oriented systems (plus the base types)

1. **S3** is a very casual system. It has no formal definition of classes. S3 implements a style of OO programming called generic-function OO.

   - **Generic-function OO** - a special type of function called a generic function decides which method to call.

Example:	`drawRect(canvas, 'blue')`
Langauge:	R

   - **Message-passing OO** - messages (methods) are sent to objects and the object determines which function to call.

Example:	`canvas.drawRect('blue')`
Langauge:	Java, C++, and C#

2. **S4** works similarly to S3, but is more formal. There are two major differences to S3.

   - S4 has formal class definitions, which describe the representation and inheritance for each class, and has special helper functions for defining generics and methods.
   - S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.

3. **Reference classes**, called RC for short, are quite different from S3 and S4.

   - RC implements message-passing OO, so methods belong to classes, not functions.
   - `$` is used to separate objects and methods, so method calls look like `canvas$drawRect('blue')`.

C STRUCTURE

- **Underlying every R object is a C structure (or struct) that describes how that object is stored in memory.**
- The struct includes the contents of the object, the information needed for memory management and **a type**.

```
typeof()  # determines an object's base type
```

- The **"Data structures"** section explains the most common base types (atomic vectors and lists), but base types also encompass functions, environments, and other more exotic objects likes names, calls, and promises.

- To see if an object is a pure base type, (i.e., it doesn't also have S3, S4, or RC behavior), check that `is.object(x)` returns FALSE.

S3

- S3 is R's first and simplest OO system. It is the only OO system used in the base and stats package.
- In S3, methods belong to functions, called generic functions, or generics for short. S3 methods do not belong to objects or classes.
- Given a class, the job of an S3 generic is to call the right S3 method. You can recognise S3 methods by their names, which look like `generic.class()`.

  For example, the Date method for the `mean()` generic is called `mean.Date()`

  This is the reason that most modern style guides discourage the use of . in function names, it makes them look like S3 methods.
- See all methods that belong to a generic :

```
methods('mean')

#> mean.Date
#> mean.default
#> mean.difftime
```

- List all generics that have a method for a given class :

```
methods(class = 'Date')
```

- S3 objects are usually built on top of lists, or atomic vectors with attributes. Factor and data frame are S3 class.

Check if an object is a S3 object	`is.object(x) & !isS4(x)` or `pryr::otype()`
Check if inherits from a specific class	`inherits(x, 'classname')`
Determine class of any object	`class(x)`

Created by Arianne Colton and Sean Chen
data.scientist.info@gmail.com

Based on content from
“Advanced R” by Hadley Wickham

Updated: January 15, 2016