1. **Four Special Environments**

   - **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()`.
   - **Base environment**, access with `baseenv()`, is the environment of the base package. Its parent is the empty environment.
   - **Empty environment**, access with `emptyenv()`, is the ultimate ancestor of all environments, and the only environment without a parent. Empty environments contain nothing.
   - **Current environment**, access with `environment()`.

2. **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.

3. **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.

4. **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.

---

### SEARCH PATH

**What is the Search Path?**

An R internal mechanism 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.

### 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**

```r
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".
```

### 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.

---

**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.

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

**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 in developing functions that aid interactive data analysis.

---

**Figure 2. Function Environment**

- The deep environment is the top-level 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()`.
- The base environment is the environment of the base package. Its parent is the empty environment.
- The empty environment is the ultimate ancestor of all environments, and the only environment without a parent. Empty environments contain nothing.
- 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.
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(f()) 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. (e.g. +, 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)

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.

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.

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 (...) # primitive

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

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

- All user-created infix functions must start and end with %

```
%k+% <- function(a, b) { # return true or false
  return
}
```

- You can also create infix functions where the function name comes in between its arguments, like ‘+’ or ‘%’

```
%1+% <- function(a, b) { # return true or false
  return
}
```

- All user-created infix functions must start and end with %

```
%k+% <- function(a, b) { # return true or false
  return
}
```

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

```
%1+% <- function(a, b) if (!is.null(a)) a else b
```

```
%1+% <- function(a, b) if (!is.null(a)) a else b
```

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.

```
replacement <- function(x, value) {
  x[2] <- value
  return(x)
}
```

- 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.

```
pryr::address()
```

DATA STRUCTURES

2. Class: represents the object’s abstract type;
- ‘type’ of the object from R’s object-oriented programming point of view
- Access with class()

```
class()
```

```
class()
```

<table>
<thead>
<tr>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeof</td>
</tr>
<tr>
<td>strings or vector of strings</td>
</tr>
<tr>
<td>numbers or vector of numeric</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td>data.frame</td>
</tr>
</tbody>
</table>

* 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().

```
is.atomic() || is.list()
```

```
is.atomic() || is.list()
```

<table>
<thead>
<tr>
<th>Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>is.atomic()</td>
</tr>
<tr>
<td>Length</td>
</tr>
<tr>
<td>Attributes</td>
</tr>
</tbody>
</table>

FACtORS

- Factors are built on top of integer vectors using two attributes:
  - class(x) <- 'factor'
  - levels(x) if defines the set of allowed values

```
class(x) <- 'factor'
levels(x)
```

```
class(x) <- 'factor'
levels(x)
```

- 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.

```
stringsAsFactors = FALSE
```

```
stringsAsFactors = FALSE
```

ATtribUtES

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

```
attr(x, 'attr') <- 'my vector'
```

```
attr(x, 'attr') <- '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
Dimensions | used to turn vectors into matrices and arrays
Class | used to implement the S3 object system
```

```
Names | a character vector giving each element a name
Dimensions | used to turn vectors into matrices and arrays
Class | used to implement the S3 object system
```
**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.

<table>
<thead>
<tr>
<th>SIMPLIFYING</th>
<th>PRESERVING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>x[1]</td>
</tr>
<tr>
<td>List</td>
<td>x[1]</td>
</tr>
<tr>
<td>Factor</td>
<td>x[1:4, drop = T] x[1:4]</td>
</tr>
<tr>
<td>Array</td>
<td>x[1, ] or x[1, 1] x[, 1] x[, drop = F] x[1,]</td>
</tr>
<tr>
<td>Data frame</td>
<td>x[1], or x[1:2] x[, 1] x[,] drop = F x[1]</td>
</tr>
</tbody>
</table>

- **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.

<table>
<thead>
<tr>
<th>DATA.FRAMES SUBSETTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LIST SUBSETTING</th>
<th>Matrix Subsetting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>df1[c('col1', 'col2')]</code></td>
<td><code>df1[c('col1'), 'col2']</code></td>
</tr>
</tbody>
</table>

The subsetting results are the same in this example.

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

  ```r
  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
  ```

<table>
<thead>
<tr>
<th>SUBSETTING (OPERATORS: [ ], [ , ]$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplifying subsetting returns the simplest possible data structure that can represent the output.</td>
</tr>
<tr>
<td>Preserving subsetting keeps the structure of the output the same as the input.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUT OF BOUNDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x[[ ]] ] [[ ] ] ( x[[] ] [ x[, 1] ] | x[[1]] [ x[1] [ x[1] [ x] ] [ x, 1 ]</td>
</tr>
</tbody>
</table>

- **Array**:
  - \( x[1:4, drop = T] \)
  - \( x[1:4] \)
  - \( x[, 1] \)
  - \( x[, drop = F] \)
  - \( x[1] \)
  - \( x[1] \)
  - \( x\[1\] \[ x[1] \[ x\] \[ x[1] \]

<table>
<thead>
<tr>
<th>DATA.FRAMES SUBSETTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUBSETTING OPERATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ is a useful shorthand for [ combined with character subsetting: ( x[y] ) is equivalent to ( x[['y']] ), exact = FALSE]</td>
</tr>
</tbody>
</table>

- **Operator**, **Index**, **Atomic**, **List**
  - [ ]: OBS NA list(NULL)
  - [ ]: OBS NA list(NULL)
  - [ ]: OBS Error Error
  - [ ]: OBS Error Error

- **If** the input vector is named, then the names of OBS, missing, or NULL components will be \( {} < \)\( \text{NA}\)\( > \).

<table>
<thead>
<tr>
<th>$ SUBSETTING</th>
<th>ASSIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplifying subsetting returns the simplest possible data structure that can represent the output.</td>
<td></td>
</tr>
<tr>
<td>Preserving subsetting keeps the structure of the output the same as the input.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Lookup tables</strong> (character subsetting)</td>
</tr>
</tbody>
</table>
| Character matching provides a powerful way to make lookup tables.

  ```r
  x <- c('m', 'f', 'n', 'n', 'm', 'm')
  lookup <- c(m = 'Male', f = 'Female', u = NA)
  lookup[x]  # m f f m m
  > 'Male' 'Female' 'NA' 'Female' 'Male' 'Male'

  uname(lookup[x])  # Male 'Female' NA 'Female' 'Male' 'Male'

  > 'Male' 'Female' 'NA' 'Female' 'Male' 'Male'
  ```

- **2. Matching and merging by hand** (integer subsetting)
  - Lookup table which has multiple columns of information.

  ```r
  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:**

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

  **Second method:**

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

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

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

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

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

<table>
<thead>
<tr>
<th>5. Selecting rows based on a condition** (logical subsetting)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical</strong> subsetting is probably the most commonly used technique for extracting rows out of a data frame.</td>
</tr>
</tbody>
</table>

```r
subset(df1[, 'col1'] == 'Male', 'col2')
```
## 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 of `browser()` which open an interactive debug session at an arbitrary location in the code.
- RStudio’s “Run 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 `recovery()`. This allows interactive debugging of batch code.

In batch R process —

```r
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 —

```r
load("last.dump.rda")
dump.frames()
```

### Conditional 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 programatically**:
   - `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.

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

Use `conditionMessage()` 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”.

### 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 `typeof()` and `class()`. 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”.

### 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.

   ```r
   Example: drawRect(canvas, 'blue')
   Language: R
   ```

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

   ```r
   Example: canvas.drawRect('blue')
   Language: 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.
   - `drop` 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.

```r
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 like names, calls, and promises.

### S3

- 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 Data method for the `mean()` generic is called `mean.Date()`.

This is the reason that most modern style guides discourage the use of function names, it makes them look like S3 methods.

- See all methods that belong to a generic:

```r
methods('mean')
```

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

```r
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 specific object

```r
is.object(x) & !isS3(x) or pryr::type(x)
```

### Check if inherits from a specific class

```r
inherits(x, 'classname')
```

### Determine class of any object

```r
class(x)
```