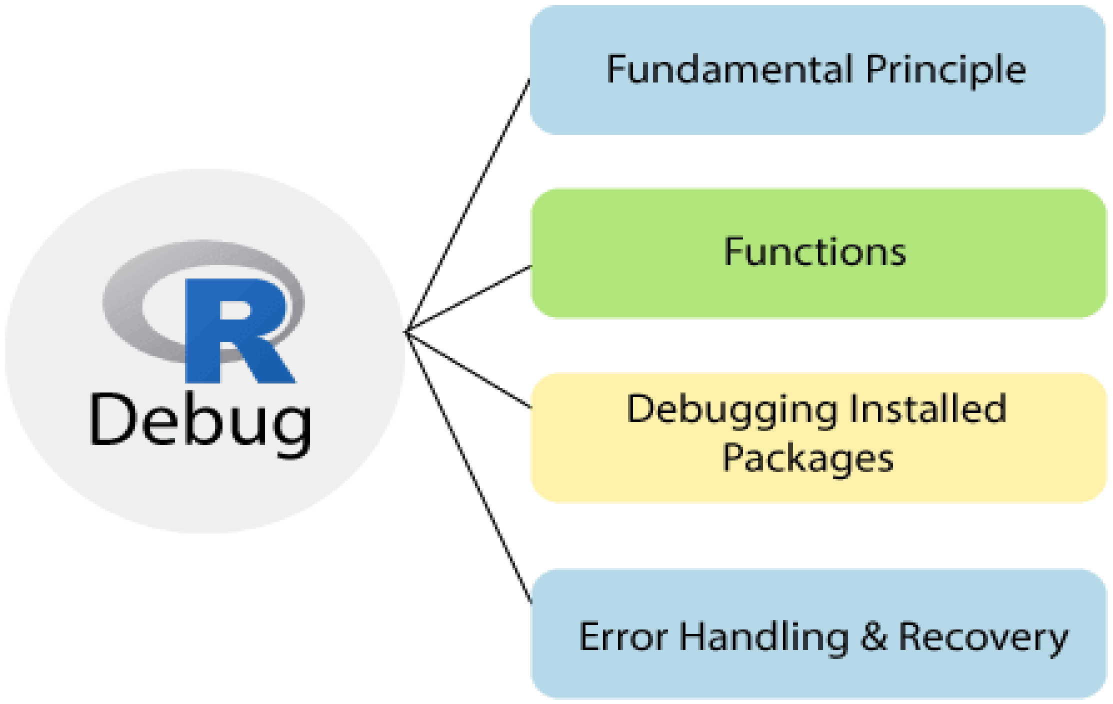
R Debug – Essential Principles and Functions

What is R Debug?

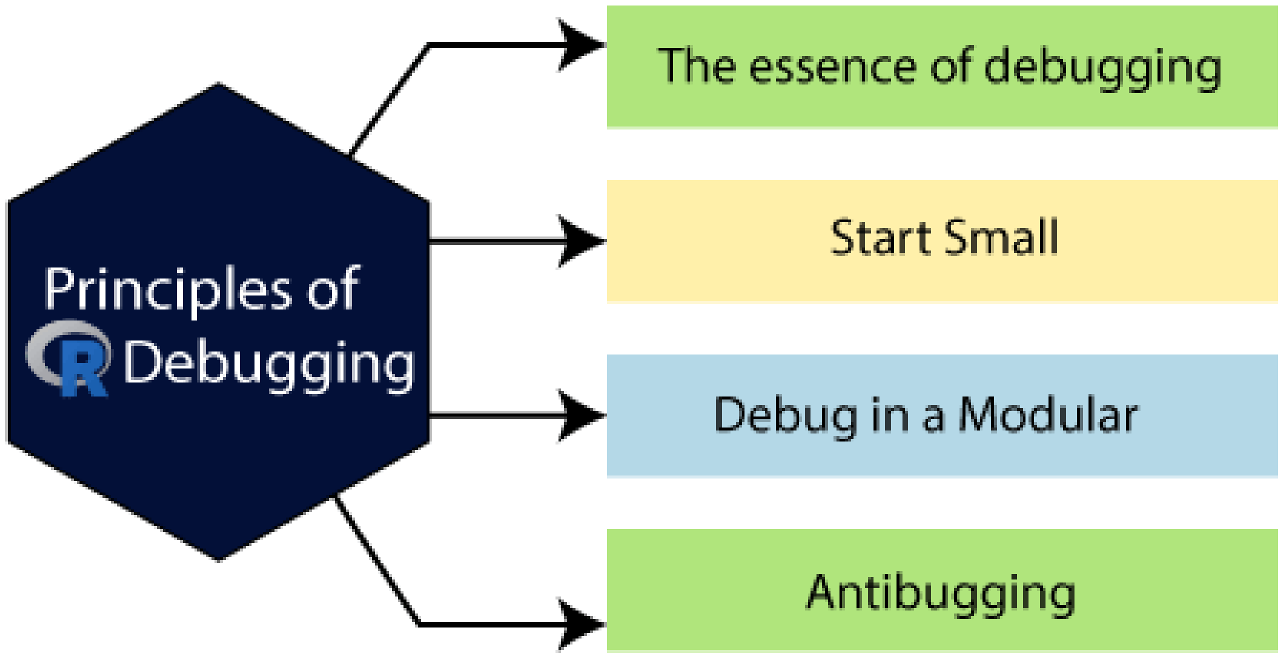
A grammatically correct program may give us incorrect results due to logical errors. In case, if such errors (i.e. bugs) occur, we need to find out why and where they occur so that you can fix them. *The procedure to identify and fix bugs is called "**debugging**".*

There are a number of R debug functions, such as:
traceback()
debug()
browser()
trace()
recover()



Fundamental Principles of R Debugging

Programmers spend more time debugging a program than actually writing it. Therefore, good debugging skills are invaluable.



1. The Essence of Debugging

The principle of confirmation: Fixing a bugging program is a process of confirming, one by one, that many things you believe to be true about code are actually true. When we find one of our assumptions is not true, we have found a clue to the location of a bug.

For example:

```
x <- y^2 + 3*g(z, 2)
w <- 28
if (w+q > 0) u <- 1 else v <- -10
```

```
> x <- y^2 + 3*g(z, 2)
Error: object 'y' not found
> w <- 28
> if (w+q > 0) u <- 1 else v <- -10
Error in w + q : non-numeric argument to
binary operator
```

2. Start Small:

Stick to small simple test cases, at least at the beginning of the R debug process. Working with large data objects may make it harder to think about the problem. Of course, we should eventually test our code in large, complicated cases, but start small.

3. Debug in a Modular

Top-Down Manner:

Most professional software developers agree that code should be written in a modular manner. Our first-level code should not be long enough with much of it consisting of functions calls. And those functions should not be too lengthy and should call another function if necessary. This makes code easier at the writing stage and also for others to understand when the time comes for the code to be extended.

We should debug in a top-down manner. Suppose we have a set the debug status of our function *f()* and it contains this line.

For example:

Y <- g(x,8)

Presently, do not call debug (g). Execute the line and see if *g()* returns the value that we expect. If it does, then we have to just avoid the time-consuming process of single-stepping through g(). If g() returns the wrong value, then now is the time to call debug(g).

4. Antibugging

If we have a section of a code in which a variable x should be positive, then we can insert this line:

Stopifnot(x>0)

If there is a bug in the code earlier that renders x equals to, say -3, the call to ***stopifnot()*** will bring things right there, with an error message like this:

Error: x > 0 is not TRUE

R Debug Functions

1. traceback()

If our code has already crashed and we want to know where the offending line is, then try *traceback().* This will show whereabouts in the code of the problem occurred.

When an R function fails, an error is printed to the screen.
Immediately after the error, **you can call traceback() to see in which function the error occurred.**
The traceback() function prints the list of functions that were called before the error occurred. **The functions are printed in reverse order**.

Example:

```
>f<-function(x) {
   r<-x-g(x)
   r

> g<-function(y) {
   r<-y*h(y)
   r
 }

> h<-function(z) {
    r<-log(z)
    if(r<10)
     r^2
    else
     r^3
 }

> f(3)
[1] -0.6208469
```

```
> f(-1)
Error in if (r < 10) r^2 else r^3 : missing value
where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced

> traceback()
3: h(y) at #2
2: g(x) at #2
1: f(-1)

> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

2. debug()

The function ***debug()* in R allows the user to step through the execution of a function, line by line.**

At any point, we can print out values of variables or produce a graph of the results within the function. **While debugging, we can simply type "c" to continue to the end of the current section of code**.

traceback() does not tell us where the error occurred in the function. In order to know which line causes the error, we will have to step through the function using debug().

After you see the "Browse[1]>" prompt, you can do different things:

- **Typing n executes the current line and prints the next one;**
- **By typing Q, we can quit the debugging;**
- **Typing 'where' tells where you are in the function call stack;**
- **By typing *ls()*, we can list all objects in the local environment.**

Typing an object name or print(<object name>) tells us current value of the object. **If your object has name n, c or Q, we have to use *print()* to see their values.**

```
>fun <- function(mu,val){
    sub <- val - mu
    sqr <- sub^2
    get <- sum(sqr)
   get
}
> set.seed(100)
>val <- rnorm(100)

> fun(1,val)
[1] 202.5615

> debug(fun)      #when executing debug()
> fun(1,val)
debugging in: fun(1, val)
debug at #1: {
    sub <- val - mu
    sqr <- sub^2
    get <- sum(sqr)
    get
}
Browse[2]>
```

```
> rnorm(5)
[1]  0.8591532 -0.8024887 -0.4919605 -0.2109543 -0.4636766

> rnorm(5)
[1]  0.2994796 -0.6054738 -0.4390121 -0.7207536  0.7808050

> set.seed(5)
> rnorm(5)
[1] -0.84085548  1.38435934 -1.25549186  0.07014277  1.71144087

> rnorm(5)
[1] -0.6029080 -0.4721664 -0.6353713 -0.2857736  0.1381082

> set.seed(5)
> rnorm(5)
[1] -0.84085548  1.38435934 -1.25549186  0.07014277  1.71144087
```

3. browser()

The R debug function ***browser()* stops the execution of a function until the user allows it to continue.**

This is **useful if we don't want to step through the complete code, line-by-line**, **but we want it to stop at a certain point so we can check out what is going on.**

Inserting a call to the browser() in a function will pause the execution of a function at the point where the browser() is called. Similar to using debug() except we can control where execution gets paused.

```
> h<-function(z) {
     browser() ## a break point inserted here
     r<-log(z)
     if(r<10)
     r^2
     else
     r^3
 }
> f(-1)
Called from: h(y)

Browse[1]> ls()
[1] "z"
Browse[1]> n
debug at #3: r <- log(z)
Browse[2]> n
debug at #4: if (r < 10) r^2 else r^3
Browse[2]> n
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
```

4. trace()

Calling *trace()* on a function allows the user to insert bits of code into a function. The **syntax for R debug function trace() is a bit strange** for first-time users. It **might be better off using debug().**

```
f <- function(a){
    x <- a-ql(a)
    x
}
ql<- function(b){
    r <- b*mn(b)
    r
}
mn<- function(p){
    r <- log(p)
    if(r<10)
        r^2
    else
        r^3
}
as.list(body(mn))
trace("mn",quote(if(is.nan(r)){browser()}),at=3,print=FALSE)
f(1)
f(-1)
```

5. recover()

When we are debugging a function, ***recover()* allows us to check variables in upper-level functions.**By typing a number in the selection, we are navigated to the function on the call stack and positioned in the browser environment.

We can use recover() as an error handler, set using *options()* (e.g.options(error=recover)).

When a function throws an error, execution is halted at the point of failure. **We can browse the function calls and examine the environment to find the source of the problem**.

```
>trace("h",quote(if(is.nan(r)) {recover()}), at=3, print=FALSE)
```

Debugging Installed Packages in R

There are possibilities of an error stemming from an installed R package. Some of the various ways to solve this problem are:

Setting options(error = recover) and then going line by line through the code using n.

When facing complicated problems, you can have a copy of the function code with you. **Entering function name in R console will print out function code that can be copied into the text editor.** You can then edit this, load it into the global workspace and then perform debugging.

If issues are still not resolved, then download the source code. You can also use the **devtools** package and the *install(), load_all()* functions to make the process quicker.

Error Handling and Recovery

Exception or error handling is a process of response to odd events of code that interrupts the flow of code. In general, the scope for **the exception handler begins with a try and ends with a catch**. **R provides the try (), and trycatch () functions for the same.**

The **try () function is the wrapper function for trycatch () that prints the error and then continues**. On the other hand, **trycatch () gives us control of the error function** and, optionally, also continues the process of the function.