Handling Errors and Conditions

Biostatistics 140.776

Conditions

R has a few types of conditions that indicate something unexpected has occurred

- error: a fatal condition; we cannot move forward with the code
- warning: an unexpected condition; we can move on but you should know about it
- message: something notable has occurred but you're probably fine

In your *own* functions, you may decide to produce an error, warning, or message when appropriate.
What do you if someone *else's* function produces a condition but you want to handle it differently?

Errors

Sometimes it's okay for errors to occur, because you can find a way around them or ignore them. In those cases, we need a way to tell R to ignore the error (or at least do something different).

```
f <- function(x) {
        y <- x + 1
        y
}
f(1)
[1] 2
```

But what about...

```
f("1")
Error in x + 1: non-numeric argument to binary operator
```

`try()`

The `try()` function is a simple way to catch an error without necessarily stopping the execution of a function.

- `try()` takes an expression as input, possibly wrapped in `{}`
- `try()` evaluates the expression; if no error occurs, it just returns the value.
- If an error occurs while evaluting the expression, `try()` returns a special object of class "try-error", along with the error message that was generated by `stop()`
- Either way, `try()` always returns something and never stops execution of R.

A Bad Function

```
bad_func <- function(x) {
        y <- x + 1
        y
}
a <- try(bad_func(1))  ## OK
b <- try(bad_func("1"))
```

try()

What does `try()` return in case there is an error?

```
b
[1] "Error in x + 1 : non-numeric argument to binary operat
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in x + 1: non-numeric argument to binary opera
```

This is an object of class `try-error`, which is something you can check for.

```
inherits(b, "try-error")
[1] TRUE
```

Using `try()`

Often, when using `for` loops, most of the inputs will be okay but some may be problematic. In these scenarios you want to finish the loop and deal with the problem later.
This loop doesn't finish.

```
input <- list(43, "a", 3)
results <- list(length(input))
for(i in seq_along(input)) {
        results[[i]] <- bad_func(input[[i]])
}
Error in x + 1: non-numeric argument to binary operator
```

Using `try()`

This loop will finish regardless of the error

```
input <- list(43, "a", 3)
results <- list(length(input))
for(i in seq_along(input)) {
        results[[i]] <- try({
                bad_func(input[[i]])
        })
}
```

Using `try()`

The result is

```
results
[[1]]
[1] 44

[[2]]
[1] "Error in x + 1 : non-numeric argument to binary operat
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in x + 1: non-numeric argument to binary opera

[[3]]
[1] 4
```

Filtering `try()` Results

Now we can deal with the errors and just treat them as `NA` values

```
for(i in seq_along(results)) {
        if(inherits(results[[i]], "try-error")) {
                results[[i]] <- NA
        }
}
results
[[1]]
[1] 44

[[2]]
[1] NA

[[3]]
[1] 4
```

Filtering `try()` Results

Another approach/paradigm

```
bad <- sapply(results, function(x) {
         inherits(x, "try-error")
})
results[!bad]
[[1]]
[1] 44

[[2]]
[1] 4
```

Using `tryCatch()`

The `tryCatch()` function is a more sophisticated version of the `try()` function. It allows for specific actions for errors, warnings, and other conditions.

- Like `try()`, `tryCatch()` takes an expression as input, possibly wrapped in `{}`
- You can add **condition handlers**, which are functions that take condition objects (error, warning)
- The condition object contains some information about what triggered the condition (i.e. a message)
- These functions handle conditions and return something (or nothing)

Another Bad Function

```
bad_func <- function(x) {
        y <- log(x + 1)
        y
}

bad_func("1")
Error in x + 1: non-numeric argument to binary operator

bad_func(-10)
Warning in log(x + 1): NaNs produced
[1] NaN
```

Using tryCatch()

We can catch the error and the warning with tryCatch().

```
input <- list(43, "a", -10, 24)
output <- lapply(input, function(x) {
        tryCatch({
                bad_func(x)
        }, error = function(e) {
                cat("error for input:", x, "\n")
                NA
        }, warning = function(w) {
                cat("warning: ", conditionMessage(w),
                    "\n")
                NA
        })
})
error for input: a
warning:  NaNs produced
```

Using tryCatch()

```
output
[[1]]
[1] 3.78419

[[2]]
[1] NA

[[3]]
[1] NA

[[4]]
[1] 3.218876
```

Cleaning Up After `tryCatch()`

The `tryCatch()` function has a `finally` argument that allows you to pass an expression for "cleaning up".

- Often when we deal with external resources, we need to open and close them
- (Bad) things can happen between opening and closing
- A common paradigm is to open a file or database connection, read some data, close the file/database connection
- If something happens in between open and close, the connection is left open
- Another example: cleaning up intermediate files that are no longer needed

Cleaning Up After `tryCatch()`

Here is an example of closing a graphics device.

```
y <- rnorm(100)
png("myplot.png")
tryCatch({
        plot(x, y)
}, error = function(e) {
        cat(conditionMessage(e), "\n")
}, finally = {
        cat("closing PNG device\n")
        dev.off()
})
object 'x' not found
closing PNG device
```

Cleaning Up After `tryCatch()`

For file connections, we sometimes open/close them directly.

```
## Create a file connection for reading
con <- file("datafile", "r")

## Read a few lines of data
d <- read_csv(con, n_max = 1000)
## Do some (error-prone) processing

## This never gets executed!
close(con)
```

Cleaning Up After `tryCatch()`

Here is an example of dealing with file connections.

```
con <- file("datafile", "r")
tryCatch({
        d <- read_csv(con, n_max = 1000)
        ## Do some (error-prone)
        ## processing
}, error = function(e) {
        cat("there was an error!\n")
}, finally = {
        ## Always gets executed
        close(con)
})
there was an error!
```

Summary

- Errors and warning can be generated by functions, but sometimes we want to handle them in a different way
- The `try()` function catches errors and returns a special object of class "try-error", which can be checked for.
- The `tryCatch()` function catches errors, warnings, and other conditions and can be used to clean up external resources
- Using `try()` or `tryCatch()` can induce a performance penalty so don't use them unless it's necessary