Error catching can be hard to catch at times (no pun intended). If you're not used to error handling, this short post might help you do it elegantly.

There are many posts about error handling in R (and in fact the examples in the `purrr` package documentation are not bad either). In this sense, this post is not original.

However, I do demonstrate two approaches: both the base-R approach (`tryCatch`) and the `purrr` approach (`safely` and `possibly`). The post contains a concise summary of the two methods, with a very simple example.

In this post I'm assuming you're familiar with the basic concepts of functional programming.

A simple example for a function with errors

Let's analyze a very simple function which divides `number` by 2. The function should return an error if its input is not an actual number, otherwise it will return `number/2`. This function looks like this:

```
divide2 <- function(number){

  # make sure the input is numeric (otherwise yield an error)
  if (!is.numeric(number)) {
    stop(paste(number, "is not a number!"))
  }

  # everything is fine, return the result
  number/2
}

divide2(10)
## [1] 5
divide2(pi)
## [1] 1.570796
```

But trying a string will yield an error:

```
divide2("foobar")
Error in divide2("foobar") : foobar is not a number!
```

Where is the problem?

What happens if we have a dataframe (or a list, or any other object for that matter) and we want to try to divide numbers within that object? Invalid values might crash our attempt completely.

For example, this loops through two values (10 and \(\pi\)) and yields their division by 2.

```
library(purrr)
map(list(10, pi), divide2)
## [[1]]
## [1] 5
##
## [[2]]
## [1] 1.570796
```

But the next version fails completely, because it tries to loop through “foobar” which cannot be divided.

```
map(list("foobar", 10, pi), divide2)
Error in .f(.x[[i]], ...) : foobar is not a number!
```

No matter where we place the invalid value “foobar”, it will fail our code completely, and we get nothing.

The solution: a `safely`/`possibly` wrapper

Fortunately, there are very simple wrappers which can help us handle the errors elegantly. These are the two functions from the `purrr` package: `safely` and `possibly`.

We’ll first demonstrate the simpler version, `possibly`. It allows us to replace errors with a chosen value. Since we expect a number, let’s replace errors with `NA_real_` (which is like saying an unknown value which is a real number).

```
possibly_divide2 <- possibly(divide2, otherwise = NA_real_, quiet =
TRUE)
```

The `quiet = TRUE` argument is just so errors will not be printed during the loop. Now we are ready to try our error safe variation.

```
possibly_out <- map(list("foobar", 10, pi), possibly_divide2)
possibly_out
## [[1]]
## [1] NA
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 1.570796
```

We can also use `unlist` to turn the output into a simple vector, i.e. `unlist(possibly_out)` will yield the vector *NA, 5, 1.5707963*.

The `safely` variation has some more strength into it, since it also provides the error messages. The output is slightly more complex though.

```
safely_divide2 <- safely(.f = divide2, otherwise = NA_real_, quiet = T)
safely_out <- map(list("foobar", 10, pi), safely_divide2)
safely_out
## [[1]]
## [[1]]$result
## [1] NA
##
## [[1]]$error
##
##
##
```

```
## [[2]]
## [[2]]$result
## [1] 5
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## [1] 1.570796
##
## [[3]]$error
## NULL
```

Following the approach suggested in the documentation of `safely` (in the examples section), we can use `transpose()` and `simplify_all()` to arrange the output.

```
simply_safe <- safely_out %>%
  transpose() %>%
  simplify_all()

simply_safe$result  # for the output values
## [1]       NA 5.000000 1.570796
simply_safe$error  # for the error messages
## [[1]]
##
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

For comparison's sake, how would you `tryCatch` it?

The function `tryCatch` is a base-R approach for error handling. The concept is similar but the syntax is different. Here is an example of how to build a safe `divide2` function with `tryCatch`:

```
try_divide2 <- function(number){
  tryCatch(expr = {
    divide2(number)
  },
  error = function(e){
    NA_real_
  }
  )
}

map(list("foobar", pi, 10), try_divide2)
## [[1]]
## [1] NA
```

```
##
## [[2]]
## [1] 1.570796
##
## [[3]]
## [1] 5
```

As said - same output, different syntax. Choose your approach. As an appetizer, the same works with base-R functional programming as well. For example, with `lapply` it will look like this:

```
lapply(list("foobar", pi, 10), try_divide2)
## [[1]]
## [1] NA
##
## [[2]]
## [1] 1.570796
##
## [[3]]
## [1] 5
```

Bonus: a benchmark

Before going off on your merry, error handled way, I also provide a short comparison between `safely`, `possibly`, and `tryCatch`.

Let's assume a data set with 20% errors. For simplicity, we'll use a modified `log` instead of our `divide2` (similar to the example provided in `purrr`'s documentation). It makes sense to fail `log` in an all numeric vector (yield an error if there is a negative value). Since a negative log is `NaN` (not an error but rather a warning) I'm creating an `error_prone_log` function.

```
library(tibble)
library(dplyr)
library(microbenchmark)
library(ggplot2)

error_prone_log <- function(x){
  if (x < 0){
    stop("Negative x")
  }

  log(x)
}

safe_log <- safely(error_prone_log, otherwise = NaN, quiet = T)
possible_log <- possibly(error_prone_log, otherwise = NaN, quiet = T)
tryCatch_log <- function(x){
  tryCatch(expr = error_prone_log(x),
           error = function(e){
             NaN
             })
}
```

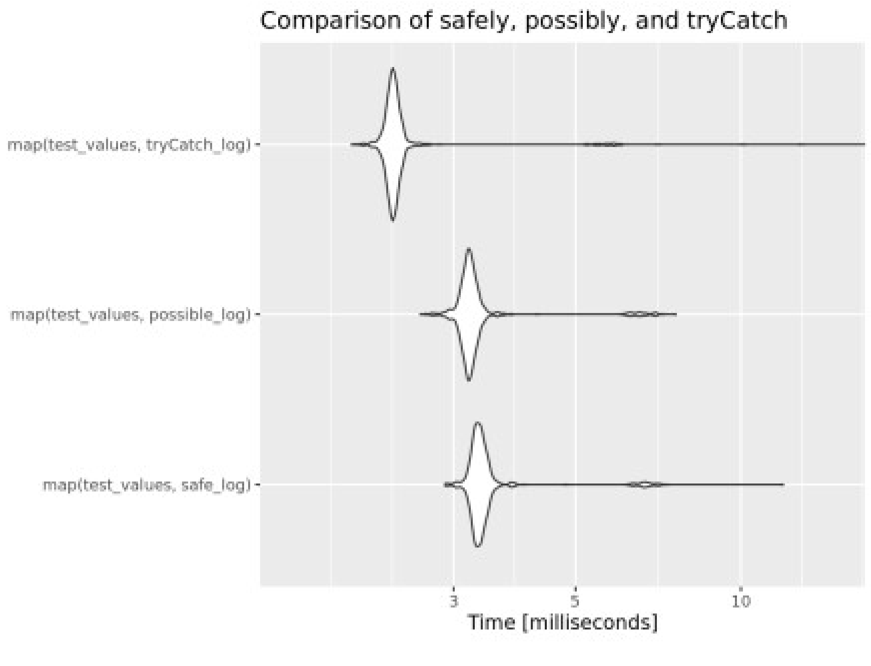
```
set.seed(42)
test_values <- sample(
  c(runif(n = 80, min = 0, max = 100),
    runif(n = 20, min = -100, max = 0)),
  size = 100, replace = FALSE)

bench_results <- microbenchmark(map(test_values, safe_log),
                                map(test_values, possible_log),
                                map(test_values, tryCatch_log),
                                times = 1000)

gt::gt(summary(bench_results)) %>%
  gt::fmt_number(columns = 2:7, decimals = 2)
```

expr	min	lq	mean	median	uq	max	neval
map(test_values, safe_log)	2.89	3.28	3.50	3.34	3.42	11.94	1000
map(test_values, possible_log)	2.62	3.14	3.35	3.21	3.28	7.62	1000
map(test_values, tryCatch_log)	1.96	2.29	2.52	2.33	2.38	77.12	1000

```
autoplot(bench_results) +
  ggtitle("Comparison of safely, possibly, and tryCatch") +
  coord_flip(ylim = c(1.5, 15))
```



As can be seen from the benchmark's results, there is no doubt about it: `tryCatch` is the fastest. The `purrr` functions `safely` and `possibly` take longer to run. In my opinion though they are slightly more convenient in terms of syntax.

In addition, the `safely` variation allows us to retrieve the error messages conveniently for later examination. Again, this capability comes with a price when compared to `tryCatch`, but it is roughly the same run time when compared to `possibly`.

Conclusion

When you have a long looping process which is prone to errors, for example a pricey web API call or a really large data set, you should aim to catch and handle errors gracefully instead of hoping for the best.

If you really need to be efficient, it’s probably worth while to `tryCatch`, and if your looking more for ease of use and code readability you should use `possibly`. In case you need to examine the error messages more thoroughly, use `safely`.

Good luck hunting down the errors!