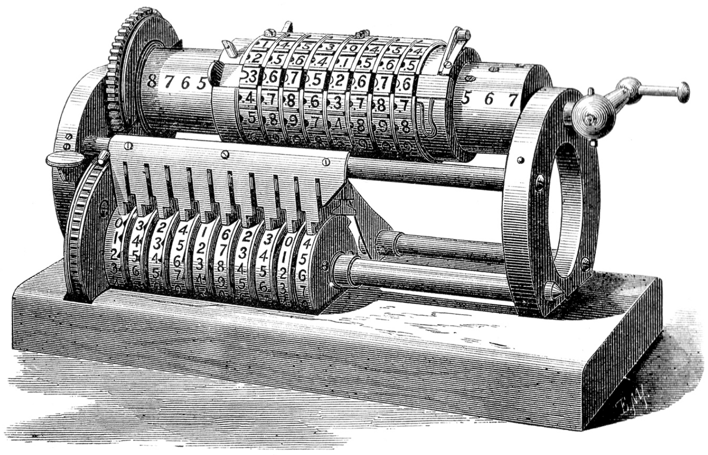
Programming in R

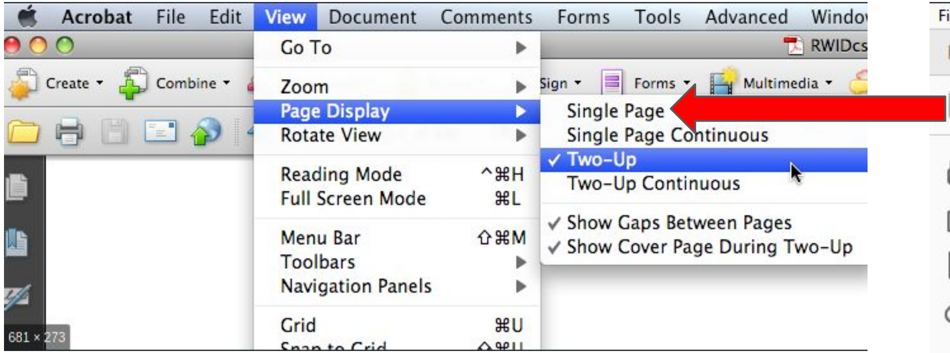
Binder

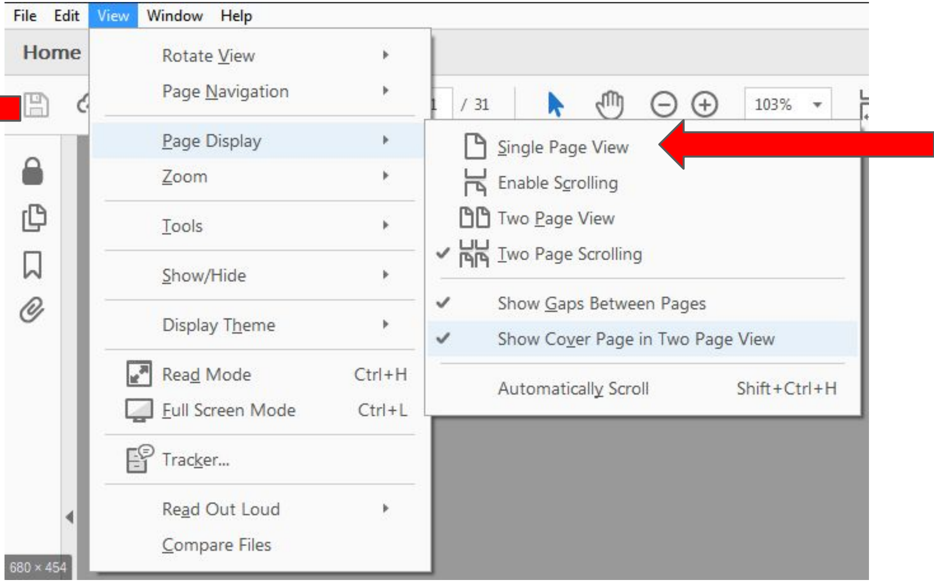


Unit 4: Debugging and Modular Software

About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Debugging and Modular Software

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "R" Track: Conditions and Errors
- "Dev" Track: Modular Programming

R Track

Conditions, Errors

Conditions and Errors

- Sometimes your function can not do what it is supposed to do
  - bad input
  - ran out of memory
  - there is no solution
  - ...
- Three ways to handle this
  1. return a value that should be what is wanted in most cases but give a *warning*
  2. return a value indicating the problem
  3. signal an error-condition (i.e. "throw an error")

Conditions and Errors

- **Warnings:** Sometimes you want to inform the user about something that is probably wrong, but you don't want to interrupt the computation
    - your function gives a result that is probably not what is wanted, e.g. NA
    - the function is superseded by a different function, but you don't want to break the code of someone who relies on the old function
    - the way the function was called may give a bad result, consider e.g.:

```
> lm(Species ~ ., data = iris)

Call:
lm(formula = Species ~ ., data = iris)

Coefficients:
 (Intercept)  Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
     1.18650      -0.11191      -0.04008       0.22865       0.60925

Warning messages:
1: In model.response(mf, "numeric") :
  using type = "numeric" with a factor response will be ignored
2: In Ops.factor(y, z$residuals) : '-' not meaningful for factors
```

Conditions and Errors

- **Warnings:** Sometimes you want to inform the user about something that is probably wrong, but you don't want to interrupt the computation
  - your function gives a result that is probably not what is wanted, e.g. NA
  - the function is superseded by a different function, but you don't want to break the code of someone who relies on the old function
  - the way the function was called may give a bad result
- In this case give a "warning":

```
> warning("this is a message")
Warning message:
this is a message
```

Conditions and Errors

- **Warnings:** Sometimes you want to inform the user about something that is probably wrong, but you don't want to interrupt the computation
    - your function gives a result that is probably not what is wanted, e.g. NA
    - the function is superseded by a different function, but you don't want to break the code of someone who relies on the old function
    - the way the function was called may give a bad result
- In this case give a "warning"
- Warnings can be ignored by using suppressWarnings()

```
> suppressWarnings(lm(Species ~ ., data = iris))

Call:
lm(formula = Species ~ ., data = iris)

Coefficients:
 (Intercept)  Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
     1.18650      -0.11191      -0.04008       0.22865       0.60925
```

(No warning is given)

Conditions and Errors

- **Warnings:** Sometimes you want to inform the user about something that is probably wrong, but you don't want to interrupt the computation
  - your function gives a result that is probably not what is wanted, e.g. NA
  - the function is superseded by a different function, but you don't want to break the code of someone who relies on the old function
  - the way the function was called may give a bad result
- In this case give a "warning"
- Warnings can be ignored by using suppressWarnings()
- The user can control how warnings are handled using options():
  - options(warn = -1) --> all warnings are ignored
  - options(warn = 0) --> warnings are returned at the *end* of a computation (default)
  - options(warn = 1) --> warnings are printed immediately
  - options(warn = 2) --> warnings are treated like errors (see next slides)

Conditions and Errors

- Weaker form of warning: **message**

```
> message("this is a message")
this is a message
```

- This should be used in place of `cat()` or `print()` to inform the user about things that may be important
- Don't overdo it with messages. It is best to make messages optional using a configuration argument, or use packages like the `lgr` package that allows configuration of output logging.
- Messages can be avoided with suppressMessages()
- there is no equivalent to options(warn=) for messages

Conditions and Errors

Returning a value indicating a problem

- Usually a bad idea!

  ```
  > res <- myfun(input)
  ```

  did it work? Suppose "NULL" indicates a problem -- now we have to check:

  ```
  > if (is.null(res)) {
  +  ...
  ```

- Returning NULL or an empty vector may make sense if it is a natural representation of "no solution"

  ```
  > findAllAppointments("2020-05-13")
  [1] "Meeting 12:00"    "Meeting 14:00"    "Conference 18:00"
  > findAllAppointments("2020-05-14")
  character(0)
  ```

Conditions and Errors

Signaling an Error-Condition

- ...using the `stop()` function
- or using functions that call stop internally, such as checkmate assertXxx(), or stopifnot()

Conditions and Errors

Signaling an Error-Condition

Example:

```r
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

Signaling an Error-Condition

Example:

- Normal control flow:

```
> f1(c(1, 2, 3))
Before calling f2
Before calling f3
```

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

Signaling an Error-Condition

Example:

- Normal control flow:

```
> f1(c(1, 2, 3))
Before calling f2
Before calling f3
After calling f3
After calling f2
[1] 8
```

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Return 7

Return 6

Conditions and Errors

Signaling an Error-Condition

Example:

- Errored control flow:

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
```

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

Signaling an Error-Condition

Example:

- Errored control flow:

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
Error in f3(arg) : arg must be numeric
```

Error

Error

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

- In the simplest case, errors raised with `stop()` (or functions that call stop) will "unwind" the call stack until they reach the interactive session (or just exit the program if it is run with `Rscript`).
- Can *catch* errors with `tryCatch()` or `try()`.
- Can make sure something gets executed even if error was given using `on.exit()`.

Conditions and Errors

on.exit

- give a command to be executed when the function is being exited

```
> f1(c(1, 2, 3))
Before calling f2
Before calling f3
```

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  on.exit(cat("exiting f1\n"))
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  on.exit(cat("exiting f2\n"))
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

on.exit

- give a command to be executed when the function is being exited

```
> f1(c(1, 2, 3))
Before calling f2
Before calling f3
After calling f3
exiting f2
```

Executing this on exit

Return 6

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  on.exit(cat("exiting f1\n"))
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  on.exit(cat("exiting f2\n"))
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

on.exit

- give a command to be executed when the function is being exited

```
> f1(c(1, 2, 3))
Before calling f2
Before calling f3
After calling f3
exiting f2
After calling f2
exiting f1
[1] 8
```

Return 7

Return 6

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  on.exit(cat("exiting f1\n"))
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  on.exit(cat("exiting f2\n"))
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

on.exit

- give a command to be executed when the function is being exited

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  on.exit(cat("exiting f1\n"))
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  on.exit(cat("exiting f2\n"))
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
```

Conditions and Errors

on.exit

- give a command to be executed when the function is being exited

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
Error in f3(arg) : arg must be numeric
exiting f2
```

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  on.exit(cat("exiting f1\n"))
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  on.exit(cat("exiting f2\n"))
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
```

Executing this on exit

Print error message

Conditions and Errors

on.exit

- give a command to be executed when the function is being exited

```r
f1 <- function(arg) {
  cat("Before calling f2\n")
  on.exit(cat("exiting f1\n"))
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  on.exit(cat("exiting f2\n"))
  res <- f3(arg) + 1
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
Error in f3(arg) : arg must be numeric
exiting f2
exiting f1
```

Error

Error

Conditions and Errors

on.exit

- give a command to be executed when
  the function is being exited
- ... even if being exited because of an error, or because the user pressed Ctrl-C, or pressed the "stop" icon in RStudio
    - so don't spend too much time in on.exit, or the user will press Ctrl-C again and interrupt this
- this may be useful to "clean up" and to avoid leaving objects in a half-broken state*

* but be careful that, in theory, your cleanup handler could be interrupted by Ctrl-C. I wouldn't rely on suspendInterrupts() saving you here! If you absolutely must perform the action in on.exit() to avoid losing data then things get very tricky.

Conditions and Errors

on.exit

- give a command to be executed when
  the function is being exited
- ... even if being exited because of an error, or because the user pressed Ctrl-C, or pressed the "stop" icon in RStudio
  - so don't spend too much time in on.exit, or the user will press Ctrl-C again and interrupt this
- this may be useful to "clean up" and to avoid leaving objects in a half-broken state*
- subsequent calls to on.exit within the same function *replace* the old on.exit handler, unless `add = TRUE` is given; see ?on.exit

* but be careful that, in theory, your cleanup handler could be interrupted by Ctrl-C. I wouldn't rely on suspendInterrupts() saving you here! If you absolutely must perform the action in on.exit() to avoid losing data then things get very tricky.

Conditions and Errors

tryCatch

Gets executed

If an error of this class gets thrown...

Then this function (the "condition handler") gets executed

```r
tryCatch(
  <expression>,
  <conditionclass1> = function(cond) {
    <react to condition>
  },
  <conditionclass2> = function(cond) {
    <react to condition>
  },
  finally = <finally expression>
)
```

This gets executed in any case, even for errors

Conditions and Errors

tryCatch

Execute this

On condition "error" do this (i.e. return 0)

```r
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- tryCatch(
    f3(arg) + 1,
    error = function(e) {
      cat("Error caught!\n")
      0
    }
  )
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Remember we are using an anonymous function here

Conditions and Errors

tryCatch

no error: as before

```
> f1(c(1, 2, 3))
Before calling f2
Before calling f3
After calling f3
After calling f2
[1] 8
```

Return 7

Return 6

```r
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- tryCatch(
    f3(arg) + 1,
    error = function(e) {
      cat("Error caught!\n")
      0
    }
  )
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Conditions and Errors

tryCatch

with error:

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
Error caught!
```

```
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- tryCatch(
    f3(arg) + 1,
    error = function(e) {
      cat("Error caught!\n")
      0
    }
  )
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

The error-handler returns 0, so `res` gets the value 0

Error

Conditions and Errors

tryCatch

with error:

```
> f1(c("a", "b", "c"))
Before calling f2
Before calling f3
Error caught!
After calling f3
After calling f2
[1] 1
```

```r
f1 <- function(arg) {
  cat("Before calling f2\n")
  res <- f2(arg) + 1
  cat("After calling f2\n")
  res
}

f2 <- function(arg) {
  cat("Before calling f3\n")
  res <- tryCatch(
    f3(arg) + 1,
    error = function(e) {
      cat("Error caught!\n")
      0
    }
  )
  cat("After calling f3\n")
  res
}

f3 <- function(arg) {
  if (!is.numeric(arg))
    stop("arg must be numeric")
  sum(arg)
}
```

Return 0

Error

Conditions and Errors

tryCatch

- the "value" of tryCatch is the value of the *expression* if no error happens, and the return-value of the error handling function if an error happens.

```
> res <- tryCatch(1 + 1, error = function(e) 3)
> res
[1] 2
> res <- tryCatch(stop(), error = function(e) 3)
> res
[1] 3
```

Conditions and Errors

tryCatch

- the "value" of tryCatch is the value of the *expression* if no error happens, and the return-value of the error handling function if an error happens.
- "finally" happens even if the error is not caught

```
> tryCatch({ cat("beginning\n") ; stop("error") }, finally = cat("end\n"))
beginning
Error in tryCatchList(expr, classes, parentenv, handlers) : error
end
```

Conditions and Errors

tryCatch

- the "value" of tryCatch is the value of the *expression* if no error happens, and the return-value of the error handling function if an error happens.
- "finally" happens even if the error is not caught
- the error handling function gets a "condition" argument that can be inspected with conditionMessage() and conditionCall()

```
> tryCatch(
+     stop("this is an error"),
+     error = function(cond) {
+       cat("The message was:",
+         conditionMessage(cond),
+         "\n")
+     }
+ )
The message was: this is an error
```

Conditions and Errors

tryCatch

- the "value" of tryCatch is the value of the *expression* if no error happens, and the return-value of the error handling function if an error happens.
- "finally" happens even if the error is not caught
- the error handling function gets a "condition" argument that can be inspected with conditionMessage() and conditionCall()
- tryCatch can react to other signals (warning, message). If tryCatch catches these, then no warning message is given (unless warning() is called again).

```
> tryCatch(
+     warning("this is a warning"),
+     error = function(cond) "error",
+     warning = function(cond) "warning",
+     message = function(cond) "message"
+ )
[1] "warning"
```

Conditions and Errors

tryCatch

- you can define custom error-classes using errorCondition(message, class =)
- tryCatch can catch these specifically

```
> tryCatch(
+   stop("error"),
+   MyError = function(cond) "my special error",
+   error = function(cond) "normal error"
+ )
[1] "normal error"
```

vs.

```
> tryCatch(
+   stop(errorCondition("error", class = "MyError")),
+   MyError = function(cond) "my special error",
+   error = function(cond) "normal error"
+ )
[1] "my special error"
```

If all you want to do is catch your special error, then this handler is not necessary btw.

Conditions and Errors

tryCatch

- you can define custom error-classes using errorCondition(message, class =)
- tryCatch can catch these specifically
- This can be used to react differently to different kinds of errors

Conditions and Errors

tryCatch

- you can define custom error-classes using errorCondition(message, class =)
- tryCatch can catch these specifically
- This can be used to react differently to different kinds of errors
- The user pressing Ctrl-C / the stop-icon in RStudio gives the "interrupt" condition. tryCatch(..., interrupt = function(cond) { ... }) can react to this!
    - This is usually a bad idea; if the user wants to quit, the program should quit. You should usually use on.exit() for cleaning up.
    - in any case be careful what you do there, the user may press Ctrl-C again in quick succession and interrupt you here, too.

Conditions and Errors

tryCatch

- tryCatch messes up debugging! **If all you want to do is modify the error message, then use withCallingHandlers instead of tryCatch**.

```
> withCallingHandlers(
+   stop("some error"),
+   error = function(cond) {
+     stop(sprintf("Kind user, it is with utmost deference that I report to you:\n%s",
+       conditionMessage(cond)))
+   })
Error in h(simpleError(msg, call)) :
  Kind user, it is with utmost deference that I report to you:
some error
```

Conditions and Errors

try()

- A short version of tryCatch()
  - returns the result of a computation if successful
  - returns a "try-error"-object if an error is thrown.

```
> res <- try(1 + 2)
> res
[1] 3
```

No error => normal result

This is just the error message print-out.
The "res" was still assigned...

```
> res <- try(1 + UndefinedVar)
Error in try(1 + UndefinedVar) : object 'UndefinedVar' not found
> res
[1] "Error in try(1 + UndefinedVar) : object 'UndefinedVar' not found\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in doTryCatch(return(expr), name, parentenv, handler): object 'UndefinedVar' not found>
```

... to be a "try-error" object

Conditions and Errors

try()

- A short version of tryCatch()
  - returns the result of a computation if successful
  - returns a "try-error"-object if an error is thrown.
- Use silent = TRUE to suppress error messages

error itself
is silent

same
error-object
as before

```
> res <- try(1 + UndefinedVar, silent = TRUE)
> res
[1] "Error in try(1 + UndefinedVar, silent = TRUE) : \n  object 'UndefinedVar' not found\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in doTryCatch(return(expr), name, parentenv, handler): object 'UndefinedVar' not found>
```

Conditions and Errors

try()

- A short version of tryCatch()
  - returns the result of a computation if successful
  - returns a "try-error"-object if an error is thrown.
- Use silent = TRUE to suppress error messages
- You *can* use try() and check if the result is an error by using

  ```
  if (inherits(res, "try-catch"))
  ```

  but it is really preferred to use tryCatch()
  - more flexible
  - what if the function in question wants to legitimately return a "try-error" object?

Conditions and Errors

Further Reading

- see ?tryCatch
- Advanced R chapter on this topic

What We Expect You to Know

- use stop(), warning(), message() for different signals
- ignore messages/warnings with suppressMessages / suppressWarnings
- change consequences of warnings using options(warn=)
- use tryCatch
    - to catch errors / stop()s
    - to catch custom errors created with errorCondition()
- know about try() (but preferably use tryCatch)
- use on.exit() to perform actions before leaving a function

Dev Track

Modular Programming

Caveat Emptor

Some of the things you learn in this course are "concrete" and "generally agreed upon to be true".

The following is one of these wishy-washy sections about how software probably kinda works, in the experience of the lecturer, but that other people possibly would have phrased differently or put different emphasis on. It is presented here in the hope that it is useful.

Software is Made from Building Blocks

- When you write software to solve some kind of problem, you need to "represent" (whatever that may mean) some relevant "concepts" (whatever they may be) of the problem in your code.

Example problem: What is the sum of two given numbers?

➔ Need to represent mathematical operations and objects (like numbers)
➔ R has this, so a "program" that sums two numbers is one line:

```r
function(a, b) a + b
```

Software is Made from Building Blocks

- When you write software to solve some kind of problem, you need to "represent" (whatever that may mean) some relevant "concepts" (whatever they may be) of the problem in your code.

Example problem: Is a move in tic-tac-toe a winning move, in a given situation?

➔ Need to represent a "move in tic-tac-toe", a given "situation" (a.k.a. "state"), the concept of "winning" etc.

➔ R does not have this directly, but we can build these concepts from other concepts:

  - game state ⇨ 3x3 matrix, containing "X"s and "O"s
  - move (i.e. where a player puts an "X" / "O") ⇨ string indicating column (numbered A, B, C) and row (numbered 1, 2, 3), e.g. "B1"
  - check for "winning" ⇨ check for three "X"s or "O"s in a line

```
  | A | B | C |
--+---+---+---+
1 | X |   |   |
--+---+---+---+
2 | O | X | O |
--+---+---+---+
3 |   | X | O |
--+---+---+---+
```

Software is Made from Building Blocks

- When you write software to solve some kind of problem, you need to "represent" (whatever that may mean) some relevant "concepts" (whatever they may be) of the problem in your code.

Example problem: Will someone who liked movie A statistically also like movie B?

- ➔ Need concept of "statistically" "liking" something, e.g. information about similar people in a database.
  - ➔ Need a "database", for example a file in which data is given in a suitable format
    - ➔ need a "file"
    - ➔ need an agreed format of how data is stored
  - ➔ Need a concept of "similar people"
    - ➔ Concept of "similarity" could be based on some matrix calculations
      - ➢ Need matrix arithmetic
      - ➢ ...
  - ➔ ...
- ➔ R does not have this, so we could implement it
- ➔ ... but usually we rely on building blocks, like a database package or a matrix factorization package.

Software is Made from Building Blocks

- How complex your software gets depends on what kind of building blocks you can build upon
  - R has (representations of) numbers, so makes things involving calculations easy
  - R comes with lots of functions for matrix arithmetic and statistics
  - There are many R software packages, especially related to problems in statistics, data science, machine learning etc.
- Some of the building blocks you have to build yourself
  - right now, we mostly tell you what functions to write
  - we will start giving exercises with functions that interact more (e.g. function ex02 calls ex01)
  - try to develop an awareness of how this gives rise to functionality that is made up of smaller parts
  - in the "real world" you will have to come up with what functions to write yourself!

Software is Modular

- Software is made up of / makes use of "Components" like R packages, functions, classes/objects (we meet these later), files, ...
- These components provide some kind of *abstraction* of concepts that you are working with.
  - ggplot2 package, quoting documentation: 'A system for 'declaratively' creating graphics, based on "The Grammar of Graphics"'
    - ggplot2 as a package provides many abstractions that represent different graphing operations, like `geom_point()` ("draw points"), or `geom_line()` ("draw lines").
    - the component "ggplot2" is an abstraction in itself: You can use it without needing to know how it works internally. Someone else could write a package with the same functions as ggplot2, but which draws things differently, for example in 3D.
  - lm() function, quoting documentation: 'lm is used to fit linear models.'
    - The function "lm()" is an abstraction of the process of fitting a linear model
    - The result of "lm()" is an abstraction of such a model. You can get its $coefficients, or call summary() on it etc.
  - The evaluate_submission.R-file you find in your homeworks: (quote) 'You can run the evaluate_submission.R script with Rscript evaluate_submission.R to check all results'
    - The file provides an abstraction of the concept of a correct / incorrect solution, in a way...

Good Software Components

- While files, R-packages, objects / classes are also "components", the components you have to worry most about are **functions**
- Once you start writing your own software (e.g. when analysing data for your Bachelor's / Master's thesis, internship, etc.) you will have to think about what functions to write
- How to decide about what functions to write?

Good Software Components

When is a component / abstraction useful?

- It represents something (e.g. an object, a process, a concept) that is "useful"
  - depends on your context!
- What it represents is "simple"
  - Simple: "a linear model", or "plotting points". Less simple: "plotting points when an even number of datapoints is given, rotating the picture by 90° otherwise."
  - ... but also depends on the context
- What it represents is meaningfully "orthogonal" to other abstractions at a similar level
  - I.e.: It provides functionality that could not have easily been reached with other "sibling" components
  - When a function "plotPoints(size = ...)" exists, then a function "plotPointsOfSize3()" is less useful
  - (However, sometimes functions that make the "common case" simpler, such as the "qplot()" function in ggplot2, make sense for usability reasons)
- The abstraction is not very "leaky", i.e. one doesn't usually need to know *how* it works to understand *what* it does.
  - E.g. you can use a `matrix` in R, and use matrix operations such as `%*%`, without having to think about how a matrix works in R
  - Most abstractions are leaky to some degree. The following is a demonstration of the fact that "numbers" in R are a leaky abstraction of the real numbers:

```
> 0.1 + 0.05 - 0.15
[1] 2.775558e-17
```

Good Software Components

When is a **function** useful?

- It does something that is "useful"
  - something that you need to solve your problem!
- What the function does is "simple"
  - I.e. can be explained in few sentences
  - See previous slide
- What the function does is "orthogonal" to other functions you have written
  - You shouldn't have multiple functions that do almost the same thing. Instead have one function with some function argument that control the details of the function's behaviour.
  - See previous slide regarding making the "common case" simpler.
    - But: often it is enough to make "default" function arguments! E.g. `plotData <- function(data, type = "points")`
- The abstraction provided by the function is not very "leaky"
  - See previous slide: One shouldn't need to know *how* it solves a problem

Designing Functions

You can split up pretty much any code that is more than one line into multiple functions, for example:

```
ex03FizzBuzz <- function(up.to, fizz.number, buzz.number) {
  sequence <- seq_len(up.to)
  ret <- as.character(sequence)
  fizzindices <- seq_len(floor(up.to / fizz.number)) * fizz.number
  buzzindices <- seq_len(floor(up.to / buzz.number)) * buzz.number
  ret[fizzindices] <- "Fizz"
  ret[buzzindices] <- "Buzz"
  ret[intersect(fizzindices, buzzindices)] <- "Fizz Buzz"
  ret
}
```

A

```
generateIndices <- function(up.to, fizz.number, buzz.number) {
  fizzindices <- seq_len(floor(up.to / fizz.number)) * fizz.number
  buzzindices <- seq_len(floor(up.to / buzz.number)) * buzz.number
  list(fizzindices = fizzindices, buzzindices = buzzindices)
}

ex03FizzBuzz <- function(up.to, fizz.number, buzz.number) {
  sequence <- seq_len(up.to)
  ret <- as.character(sequence)
  indices <- generateIndices(up.to, fizz.number, buzz.number)
  ret[indices$fizzindices] <- "Fizz"
  ret[indices$buzzindices] <- "Buzz"
  ret[intersect(indices$fizzindices, indices$buzzindices)] <- "Fizz Buzz"
  ret
}
```

B

```
generateIndices <- function(up.to, number) {
  seq_len(floor(up.to / number)) * number
}

ex03FizzBuzz <- function(up.to, fizz.number, buzz.number) {
  sequence <- seq_len(up.to)
  ret <- as.character(sequence)
  fizzindices <- generateIndices(up.to, fizz.number)
  buzzindices <- generateIndices(up.to, buzz.number)
  ret[fizzindices] <- "Fizz"
  ret[buzzindices] <- "Buzz"
  ret[intersect(fizzindices, buzzindices)] <- "Fizz Buzz"
  ret
}
```

In this example, "B" is preferred:

- "A" just cuts out an arbitrary part from ex03FizzBuzz and pastes it somewhere else
- "B" does something more: it uses the fact that "*a sequence of equally spaced numbers greater than 0 and smaller than 'up.to*" is needed multiple times.
- "B" avoids having to write the '`seq_len(floor(up.to..)) * number` ' code twice and therefore *simplifies* things.

Designing Functions

(Examples for) When to write a function:

1. (When needed for lapply(), vapply() & co. -- but this is arguably not what we are talking about here: anonymous functions are not really "components" in the way we mean here.)
2. You find yourself copy-pasting some code and only changing some variable names
   - ➜ Turn the code you are copy-pasting into a function: "Don't Repeat Yourself" (DRY)-principle.
   - ➜ This is example "B" in the previous slide.
3. Part of your code involves an operation that is a logical concept / operation that can easily be explained in a short sentence
   - ➜ Make this code a function, using a function name, argument names and variable names reflecting what the concept / operation is you are invoking
   - E.g. reading data from a file with a particular format, even if your code only has to read it once
   - E.g. calculating a complicated mathematical function that makes up a logical unit in your code, even if this code is only used once
4. The code / the function you are currently writing is getting complicated and could easily be explained in terms of smaller parts
   - E.g. if you have a deep nesting of loops or if-else blocks
   - E.g. if your code is substantially longer than one page on your screen
   - ➜ Try to split the code up into functions, even if 1. or 2. don't apply
   - ➜ This would look like example "A" in the previous slide (if we forget that case 2. applies!)

Designing Functions

How to write a function:

- Documentation:
  - Short sentence (or paragraph, if necessary) **describing what the function does**
  - Listing of function **arguments**, detailing (1) their **type** and (2) their **purpose**
  - **Return value type** and, if not clear from context, what the return value actually is.
- Function header
  - Function name should be descriptive and unambiguous
  - arguments with default values should usually come *after* arguments without default values
  - Try not to have too many arguments
- Function body
  - Good to start with **asserts**: These (1) make sure the function will not be called with unexpected values and fail in unexpected ways, and (2) make it clear for someone looking at the function, what input the function expects

```r
# Create an increasing vector of multiples of 'number' between 1 and 'up.to' (inclusive)
# 'up.to' (integer(1)) upper bound
# 'number' (integer(1)) multiples of which number to create.
#   Does not need to be a divider of 'up.to'
# returns a numeric vector.
generateIndices <- function(up.to, number = 2) {
  assertInt(up.to, tol = 1e-100, lower = 0)
  assertInt(number, tol = 1e-100, lower = 1)
  seq_len(floor(up.to / number)) * number
}
```

Designing Functions

How to write a function:

- Documentation:
  - Short sentence (or paragraph, if necessary) **describing what the function does**
  - Listing of function **arguments**, detailing (1) their **type** and (2) their **purpose**
  - **Return value type** and, if not clear from context, what the return value actually is.
- Function header
  - Function name should be descriptive and unambiguous
  - arguments with default values should usually come *after* arguments without default values
  - Try not to have too many arguments
- Function body
  - Good to start with **asserts**: These (1) make sure the function will not be called with unexpected values and fail in unexpected ways, and (2) make it clear for someone looking at the function, what input the function expects

```
# Create an increasing vector of multiples of 'number' between 1 and 'up.to' (inclusive)
# 'up.to' (integer(1)) upper bound
# 'number' (integer(1)) multiples of which number to create.
#   Does not need to be a divider of 'up.to'
# returns a numeric vector.
generateIndices <- function(up.to, number = 2) {
  assertInt(up.to, tol = 1e-100, lower = 0)
  assertInt(number, tol = 1e-100, lower = 1)
  seq_len(floor(up.to / number)) * number
}
```

How much and how urgently the documentation / asserts are needed depends on how large and complex the function is. E.g. one-liners may not need it:

```
isnt.null <- function(x) !is.null(x)
```

Remember from our Programming Style Section:

Your Audience has Tunnel Vision

Your code should be understandable:

- without needing to see many lines at once
  - chunk your code into reasonably sized functions
    - should ideally fit on a screen
  - and reasonably sized files
- without having to keep many things in mind at once
  - limit your "nesting depth": avoid for-loops inside if-clauses inside for-loops inside ...
  - limit the number of args of functions. The user shouldn't have to check the docs all the time.
  - limit the role fulfilled by a single function. If you can't summarize its effect in one sentence, consider splitting it up.
  - related but more general: strive for **loose coupling**, i.e. limit the interdependencies between parts of your code and the degree to which one part depends on specific implementation details of another part.

What We Expect You to Know

- Solve big problems by breaking them down into subproblems, for which you then write functions.
- Functions should fulfill a clear purpose that can (and usually should) be explained in few sentences.
- Function and argument names should be descriptive
- Function arguments and return values should be documented, in particular their expected type

(This is not something we can readily test in our homework / exam and is therefore "not examinable", but keeping this in mind will make your life easier when you have your own projects)