# Assertions, Errors, and Defensive Programming
Statistics 650/750
Week 5 Tuesday

Alex Reinhart and Christopher Genovese

26 Sep 2017

## Announcements

- Chris will have office hours TODAY 3:30 - 4:30, Justin 4:30 - 5:30
- New assignments on errors and exceptions will be posted shortly
- Please double-check that your pull requests are assigned to the TAs – there appear to be some that are not, and they will not be graded!
- (There is a backlog of a few days for grading, so be patient.)
- Please review your pull requests before you submit them, and don't `git add` files not related to your submission, like `.Rhistory` files, `.Rproj` stuff, or other random things that aren't the code you wrote. You also don't need to delete the `README` or other files from each homework: pull requests represent the *difference* between two branches, so all that matters is what's *changed*
- You can make a `.gitignore` file:

  ```
  *.pyc
  .Rhistory
  mydata/
  ```

- Remember that vignettes are multiple assignments
- Make sure that submitted text files are text files, not RTF or Word files
- Please don't mind me when I leave random comments

## Assertions

An assertion is a statement that a condition is true.

Assertions are intended to state conditions that are expected to always be true, and an assertion failure usually leads to program termination. If the assertion is true, nothing happens.

A couple quick examples:

```r
library(assertthat)

foo <- function(x) {
```

```
    assert_that(x >= 0)
    # ... do stuff with x
}
```

```
#include <cassert>

int adjust(int base, int increment) {
    assert(base >= 0);
    assert(increment % 2 == 0);
    // ...
}
```

As we have seen, assertions are a main ingredient of tests:

```
library("stringr")
test_that("string concats work correctly", {
    expect_equal(str_c("A", ""),   "A")
    expect_equal(str_c("A", "B"),  "AB")
    expect_equal(str_c("A", "BC"), "ABC")
})
```

Here the assertion is `expect_equal`. If the asserted condition is not true, the test fails.

Assertions inside functions add overhead, since the condition must be checked every time the code runs. Hence assertions are sometimes disabled once development is complete and the code is in use.

(Example: In C++, simply defining a macro `NDEBUG` disables all assertions.)

### What are assertions for?

Assertions are a debugging aid. They are used to help the programmer detect and fix bugs (conditions that should not occur) and verify that the underlying assumptions remain true.

For example, while working on a function to fit a complicated model, you may know that certain parameters must remain in a certain range if the model is fit correctly. If you add assertions, you will catch errors in your code before they give you nonsense results:

```
def fit(data, ...):

    for it in range(max_iterations):
        # iterative fitting code here
        ...

        # Plausibility check
        assert np.all(alpha >= 0), "negative alpha"
        assert np.all(theta >= 0), "negative theta"
        assert omega > 0, "Nonpositive omega"
        assert eta2 > 0, "Nonpositive eta2"
        assert sigma2 > 0, "Nonpositive sigma2"

    ...
```

In this same model fitting algorithm, you might know that the log-likelihood is guaranteed to increase at each iteration – if it does not, something is wrong with your code. This could be an assertion.

In general, if you find yourself thinking something like "I need to calculate this quantity, but I'm pretty sure this variable will always be *[something]*, so I can just do *[something else]* instead", that's an assertion which can be written into the code to catch if your assumption is wrong.

Assertions are not a substitute for error handling. They are *not*, for instance, intended to detect and handle erronous input or a bad program state that is not *due* to a bug in the program.

In `fit`, for example, we can't recover from `omega < 0`. It's impossible, so if it happens, our code is wrong. There is nothing sensible to do to fix it.

(However, in languages meant to be used interactively, like R, they are often used to catch when users call functions in incorrect ways.)

Assertions provide both run-time checking (especially during development) and also a simple documentation of the underlying assumptions.

R users can use the assertthat package; Python has the `assert` statement built in to the language, as do some other languages.

### How are assertions different from tests?

Assertions seem similar to unit tests: a unit test asserts that a function does a certain thing, right?

While this is true, we use assertions for more than just testing.

A unit test provides a function certain inputs and makes sure the outputs are correct. Unit tests exist *outside* the function, calling it in different ways to check its overall behavior.

But we can also write assertions *inside* functions, verifying that certain conditions are always true.

There is another common idiom we often see assertions use for: checking the inputs and outputs of a function to make sure they meet a *contract*.

### Design by Contract

This is a design methodology where programmers define precise and verifiable specifications for software components. The contract is useful for both debugging and documentation.

Design by contract is based on the metaphor of a legal contract between two parties defining the obligations embodied in a transaction (e.g., function call).

The main ingredients of a contract (usually at the function or class level) are

**preconditions** a condition that should be true just prior to execution

**postconditions** a condition that should be true immediately after execution

**invariants** a condition that should be true during execution

**side effects** modified state or observable interaction with the outside world

**error** error conditions that can occur

**returns** values, types, meaning returned

**guarantees** performance (time or space), validity (ACID), etc.

The first three are the most commonly used. Some languages, like Eiffel and Racket, have sophisticated built-in contract systems:

```
(define/contract (foo x y)
  (-> positive? positive? positive?)
  (+ (* x x) (* y y)))
```

This defines `foo` to have a contract that its arguments are positive and it returns a positive number. If we violate the contract, Racket tells us what code is to blame – the calling code, in this case:

```
(foo -2 3)

foo: contract violation
  expected: positive?
  given: -2
  in: the 1st argument of
      (-> positive? positive? positive?)
  contract from: (function foo)
  blaming: main
   (assuming the contract is correct)
  at: bad-code.rkt:3.18
```

You can add contracts to Python with an extra module:

```
from contracts import contract

@contract(lines='list(str)',
          returns='dict(str: (int,>=1))')
def word_count(lines):
    result = {}

    for line in lines:
        for word in line.split():
            result[word] = result.get(word, 0) + 1

    return result
```

In some implementations, new types of contracts can be defined separately and reused:

```
from contracts import contract, new_contract

@new_contract
def even(x):
    if x % 2 != 0:
        msg = 'The number %s is not even.' % x
        raise ValueError(msg)

    # do stuff
    ...

@contract(x='int,even')
```

```
def foo(x):
    pass

foo(2)
foo(3)

contracts.interface.ContractNotRespected: Breach for argument 'x' to foo().
The number 3 is not even.
checking: callable()   for value: Instance of int: 3
checking: even         for value: Instance of int: 3
checking: int,even     for value: Instance of int: 3
Variables bound in inner context:
- args: Instance of tuple: ()
- kwargs: Instance of dict: {}
```

Other languages take this to an extreme: SPARK (based on Ada) analyzes each function and tries to logically prove that it satisfies the specified contract, and will throw an error if the function won't satisfy the contract. This can be done without even running the code.

In languages without built-in contracts, like R, we can use assertions inside functions to check pre- and post-conditions when the code runs. These don't give such elegant error messages, but serve the same purpose. Many R users use assertions primarily for pre-condition checks, and the `assertthat` package is designed for this.

### A brief exercise

Suppose you have a function `shortest_path(graph, start_node, end_node)` which is intended to calculate the shortest path between two nodes in an undirected graph. Write a contract specifying the pre- and post-conditions the function must satisfy.

You can write informally, such as just saying "`graph` must be a graph object, and `start_node` must be a. . . ", instead of using a specific syntax.

Assume you have a range of useful functions like `is_graph`, `is_node`, `graph_contains_node`, and so on.

What conditions did you specify?

1. Test that the final path's edges are in the `graph`. Could have a `edge_in_graph(edge, graph)`
2. Precondition that there must be a path – or otherwise define what should happen if there is no path
3. Check that the output path is actually the shortest path; `is_shortest_path` function? May be expensive, to be done in a unit test instead
4. Precondition: `start_node` and `end_node` are in the graph
5. Postcondition: no cycles in returned path
6. If it's a directed graph, output path must follow arrows
7. Could output path as a subgraph – must also satisfy `is_graph`
8. Postcondition: path starts at `start_node` and ends at `end_node`

# Errors and Exceptions

## Why handle errors?

Your code will frequently need to handle errors – either caused by a user passing the wrong input or your code hitting an exceptional case.

- Another function passes the wrong kind of data to your function
- Algorithm fails to converge
- Couldn't open the data file
- Couldn't connect to the SQL database
- Network connection is unreliable or server timed out
- ...etc.

I've seen a lot of code like this:

```
def read_data_file(filename, max_rows, format_args):
  if not os.path.isfile(filename):
    return "File not found"

  f = open(filename, "r")

  # do stuff...
```

Or:

```
crowded_cows <- function(cows, K) {
    if (K > length(cows) || K < 1) {
        cat("K is out of range")
        return
    }

    # do stuff...

    if (there is no crowded cow) {
        cat("No crowded cows")
        return
    }
}
```

When you're working interactively in the REPL, manually calling functions to do things, this isn't a big deal. You can read the message and decide what to do.

But when you're writing a large project with many functions, all calling each other to do some complicated analysis, you shouldn't need to handle every error manually. How is a function calling the above `crowded_cows` supposed to detect which error has occurred?

Errors are part of the logic of the program, and we should be able to write code which handles errors and does specific things to handle them.

If we simply return a special value on errors, or just print a message, it's very easy to accidentally ignore an error or, worse, use the return value as though it were a real value. And there's no flexibility: If sometimes you want to log a message and sometimes you want execution to stop entirely, you have to code that logic into every function.

How can we reliably indicate error conditions and write code to deal with them?

## Errors versus assertions

Earlier we discussed using *assertions* to make claims about facts in your code, and to program defensively. But when would I use an assertion and when would I use an error? What's the semantic difference?

Errors are for unexpected conditions which could be *handled* by the calling code, which may want to perform some action to work around the error, fix it, or report it to the user.

An assertion indicates something which *must be true* if the program is functioning correctly. If an assertion is false, there's nothing to handle or recover from: the code is wrong and must be fixed. Assertions are sanity checks that things are working as expected.

To give a real-life example, suppose someone gives you directions to drive to their house. (Actual directions, not just Google Maps live instructions.) An error occurs when you can't recognize where you are and don't know what to do next. Your mental directions-following algorithm can recover from this error: maybe go back and retrace your steps, or call your friend, or check Google Maps to see where you are. This is a recoverable error.

An assertion, which your directions-following algorithm assumes is always true, is that your vehicle is on the ground, preferably on a road. If you find yourself underwater, the assertion has failed, and you are probably not getting to your friend's house today. You can't simply look on Google Maps and get directions to drive out of the lake. Your "how to get to Farmer Brown's house" instructions do not know how to deal with this case at all.

So an error is a foreseeable problem which your code can detect and potentially recover from; an assertion is something which must be true for your code to even be correct at all.

## Error handling paradigms

### Exceptions

Exceptions signal an error to be handled by code somewhere up the call stack. Exceptions have a type – there are different kinds of exceptions, and code can decide which to handle and which to pass on. You can define new kinds of exceptions for your own code.

Exceptions are available in Python, Java, C++, JavaScript, Julia, and many other languages.

Code can *catch* exceptions caused by functions they call, or functions called by those functions, and so on, and try to *recover* from the exceptions.

Consider my model-fitting function example again:

```python
def fit(data, initial_guess, max_iterations=100, ...):

    current_solution = initial_guess
    current_likelihood = likelihood(data, initial_guess)

    converged = False

    for it in range(max_iterations):
        next_step = update(data, current_solution)
```

```
        new_likelihood = likelihood(data, next_step)

        assert valid(next_step), "Solution is invalid"
        assert new_likelihood >= current_likelihood, \
            "Likelihood did not decrease"


        if sufficiently_close(current_solution, next_step):
            return next_step

        current_solution = next_step
        current_likelihood = new_likelihood

    raise ConvergenceError("Failed to converge after {} iterations".format(it))

def update(data, solution):
    delta = invert_big_matrix(solution)

    # do complicated math
    # ...

    return next_step
```

This model-fitting involves several functions:


Suppose `fit` fails to converge, and raises the `ConvergenceError`.

A raised exception causes the function to abort, and control returns to the calling function. If the calling function (`main`) does not *catch* the exception, it also aborts. If no function catches the exception, your code crashes and an error is printed:

```
Traceback (most recent call last):
  File "exception.py", line 21, in <module>
    main()
  File "exception.py", line 15, in main
    fit([], 10, 30)
  File "exception.py", line 11, in fit
    raise ConvergenceError("Failed to converge after {} iterations".format(it))
```

To catch the exception up in `main`, we use a `try` block:

```
def main():
    try:
        fit(data, -4, max_iterations=10)
```

```python
    except ConvergenceError as e:
        print(e.args)
        print(e)
        ...
        # do something clever here
```

Any exception inside the `try` block can be *caught* by the exception handler. Notice the exception handler specifies the kind of exceptions it handles. If your code can fail in multiple ways, you can define `except` clauses for each. You can also create new kinds of exceptions not built in to the language. If you don't write a handler for a specific type of exception, that exception will abort your function and proceed upwards until something *does* handle it.

What could we do here? We could imagine that, if we catch a `ConvergenceError`, `main` may want to retry the model fit with different parameters or maybe a different type of fitting algorithm. It could do this entirely automatically, without our intervention. Or it could do nothing, not catching the exception, in which case the program will crash and the user will have to do something.

Exceptions allow some clever error handling. For example, the "retrying" package wraps functions to automatically catch certain kinds of errors and retry:

```python
@retry(wait_exponential_multiplier=1000, wait_exponential_max=10000,
       stop_max_attempt_number=5, retry_on_exception=geocode_error)
def geocode(addr, cur, tract=None, max_rating=5):
    """Geocode an address following a tiered strategy.

    ...
    """
    coordinates = geopy.geocode(addr) # might fail

    ...
```

The geocoder has to call an external service (Google Maps), and if the network connection fails or Google's API goes down for a moment, we can catch the error and retry.

### Exceptions in R

R doesn't actually use exceptions. It has a system of *conditions*, which are more powerful and flexible than exceptions, except that I've never seen them used in R code. Conditions were pioneered in Common Lisp and its predecessors, which used them extensively for error handling.

The condition system can act like an exception system. To raise a condition (like throwing an exception), call `stop`. It can take an error message (or multiple things which it will `paste()` together to make an error message) as an argument:

```r
foo <- function(x) {
    if (x < 0) {
        stop(x, " is not positive")
    }

    ## would be easier with
    ## assert_that(x >= 0)
```

```
    ## but that's beside the point
}
```

There are other types of conditions that can be raised; for example, `warning` prints a warning message but does not abort the function, and `message` prints a message. You might use warning messages for things like "Model fit didn't converge to full precision" or some other case where the code can proceed anyway.

(Why use `warning` or `message` instead of just printing a message? A user can use a condition handler, like `suppressMessages`, to hide messages if they want.)

The `tryCatch` function runs a block of code, and if a condition is raised, runs the appropriate handler based on what you've provided.

```
tryCatch({
    data <- read_big_file(file)
    fit_model(data) },

    error=function(e) { (handle error) },
    file_not_found_error=function(e) { (do something) },
    convergence_error=function(e) { (do something else) }
)
```

(To learn how to define new kinds of errors in R, look at *Advanced R*'s error handling chapter.)

### Conditions

Exceptions have a weakness: the code recovering from the error (the `except` or `catch` block) is completely separate from the code that was running when the error occurred.

If function `main` calls `fit` which calls `update`, which calls `invert_big_matrix`, which throws a `SingularMatrixError` how can `main` handle the error and recover appropriately *without* knowing the details about how `fit` and `update` work?

R has a sophisticated condition handling system, stolen from Common Lisp, for handling these kinds of problems. You probably haven't seen it before – R is usually used interactively, so *you* are the condition handler. But for robust, reliable programs, you need automation.

But we can imagine there are many possible ways to handle this `SingularMatrixError`. We could

- Rescale the `data` to avoid numerical issues
- Remove variables from the `data` which are nearly colinear and might be causing this problem
- Calculate an approximate inverse
- Fall back to an alternative way of calculating the update step

A condition handler is a bit like an exception handler, except it allows the function which raised the condition to *continue running* – the handler decides what the function should do to recover.

```
invert_big_matrix <- function(mat) {
    if (invertible(mat)) {
        ## calculate big inverse with fancy algorithm
        ...

```

```
        return(inverse)
    }

    return(withRestarts(
        stop(singular_matrix_error(mat)),
        rescale_matrix=function() { invert_big_matrix(rescaled(mat)) },
        approximate_inverse=function() { approx_inverse(mat) },
        replace_with=function(replacement) { invert_big_matrix(replacement) },
        ...
    ))
}
```

If we encounter an error parsing the data, we raise a `singular_matrix_error` condition. (Conditions use R's object-oriented programming system; see the resources below to see how to define a new one.) We provide several *restarts*: possible ways of handling and recovering from the error.

The function calling `invert_big_matrix` chooses which restart should run:

```
update <- function(data, solution) {
    withCallingHandlers({
        delta <- invert_big_matrix(solution)
    },
    singular_matrix_error=function(mat) {
        invokeRestart("rescale_matrix")
    })
}
```

When `invert_big_matrix` raises the `singular_matrix_error`, notice it returns the value returned by the chosen restart, so `rescale_matrix` can return an inverse from a rescaled version.

R has default condition handlers for certain conditions. For example, `stop` normally aborts like an exception would. `message` writes its output to the console:

```
fit_model <- function(data, max_iters=100) {
    for it in 1:max_iters {
        message("Iteration ", it, " of ", max_iters)

        ## calculate stuff
        ...
    }
}

fit_model() # noisy
suppressMessages(fit_model()) # quiet
```

There is also `warning` for non-fatal errors, like convergence problems, and a similar `suppressWarnings` function to set a restart that *doesn't* display them.

## Resources

- Hadley Wickham's *Advanced R* has a chapter on conditions and debugging as well as a more detailed condition handling guide.
- Python's tutorial has a section on exception handling. (Note that some details changed since Python 2.)
- Julia's manual discusses handling and creating exceptions.