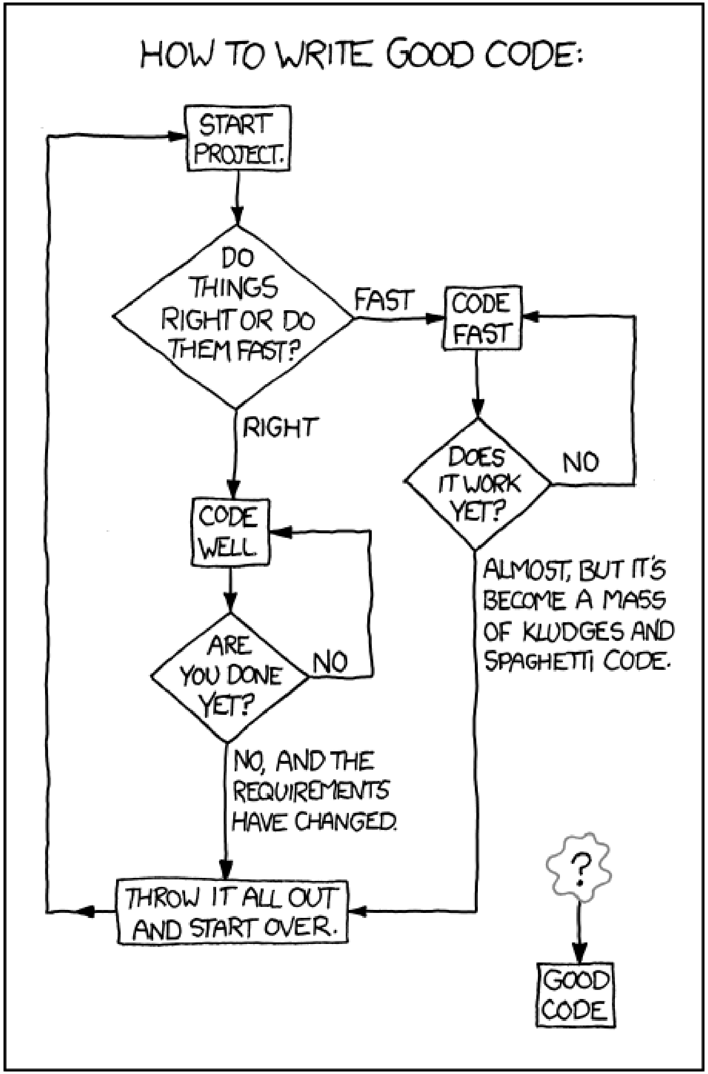
Code Quality

Best Practices for Writing Reproducible Code // part 2

Aspects of good quality code

- Readable
- Reusable
- Robust


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Source: xkcd

Code readability

- Code is for computer, comments are for humans.

Code readability

- ~~Code is for computer, comments are for humans.~~

- Use whitespace and newlines strategically.

Compare:

```
this ← function(arg1,arg2) res←arg1*arg2;return(res)
hurts ← mean(c(this(3,4),this(3,1),this(9,9))); print(hurts)
```

```
this ← function(arg1,arg2){
  res ← arg1 * arg2
  return(res)
}

hurts ← mean(
  c(
    this(3,4),
    this(3,1),
    this(9,9)
    )
  )
print(hurts)
```

Code readability

- ~~Code is for computer, comments are for humans.~~
- Use whitespace and newlines strategically.
- use descriptive names for functions and variables
    - start functions with a verb
    - make variable names *just* long enough to be meaningful

Compare:

```
for i in my_shopping_basket:
  if(test(i)) > 10:
    purch(i)
  else:
    disc(i)
```

```
for item in basket:
  if(testNecessity(item)) > 10:
    purchase(item)
  else:
    discard(item)
```

Code readability

- ~~Code is for computer, comments are for humans.~~

- Use whitespace and newlines strategically.

- use descriptive names for functions and variables

  - start functions with a verb
  - make variable names *just* long enough to be meaningful

- use a consistent style

  - consistency will make your code easier to understand and maintain
  - consult a styleguide for your language (keep conventions, and don't reinvent the wheel)

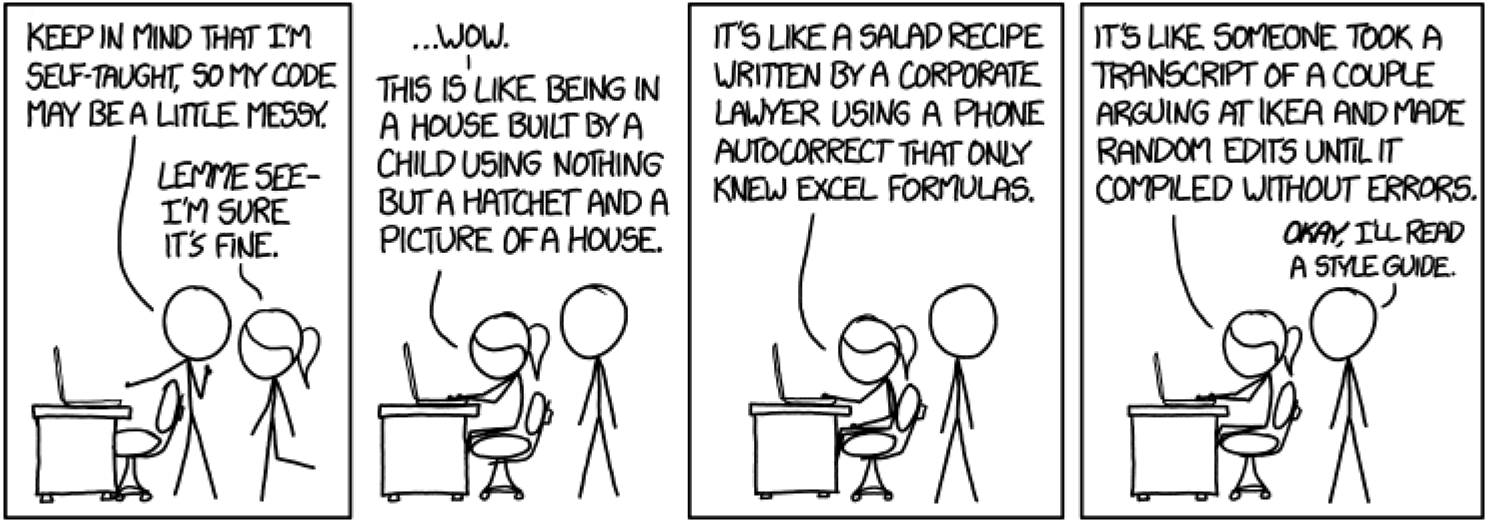
Compare:

```
myVar←original_variable+MOD(new.var)
```

```
my_var ← original_var + Modified(new_var)
```

Styleguides

- Python style manual: Pep-8
- R style manual: Tidyverse style guide


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

Source: xkcd

Your turn

Where can you improve the readability of your code?

- Run a linter (e.g. `flake8` for Python or `lintr` for R) to identify conflicts with style guides.
- If you find code that is hard to read, make a note to work on it. (Schedule time to refactor, but do not do this now!)

Tip! Use `#TODO` or `//TODO` (depending on your comment marker) to easily find these spots later on. Many IDEs extract these into a task list!

Code reusability

- Less code written, more work done
- Writing a tool while doing your analysis
- Stop reinventing the wheel!

Code reusability: some guidelines

- Separate code and data: data is specific, code need not be
    - consider using a config file for project-specific (meta)data
    - but DO hard-code unchanging variables, e.g. `gravity = 9.80665`, **once**.
- Do One Thing (and do it well)
    - One function for one purpose
    - One class for one purpose
    - One script for one purpose (no copy-pasting to recycle it!)
- Don't Repeat Yourself: use functions
    - Write routines in functions, i.e., code you reuse often
    - Identify potential functions by action: functions perform tasks (e.g. sorting, plotting, saving a file, transform data…)

Code reusability through functions

Functions are smaller code units reponsible of one task.

- Functions are meant to be reused
- Functions accept arguments (though they may also be empty!)
- What arguments a function accept is defined by its parameters

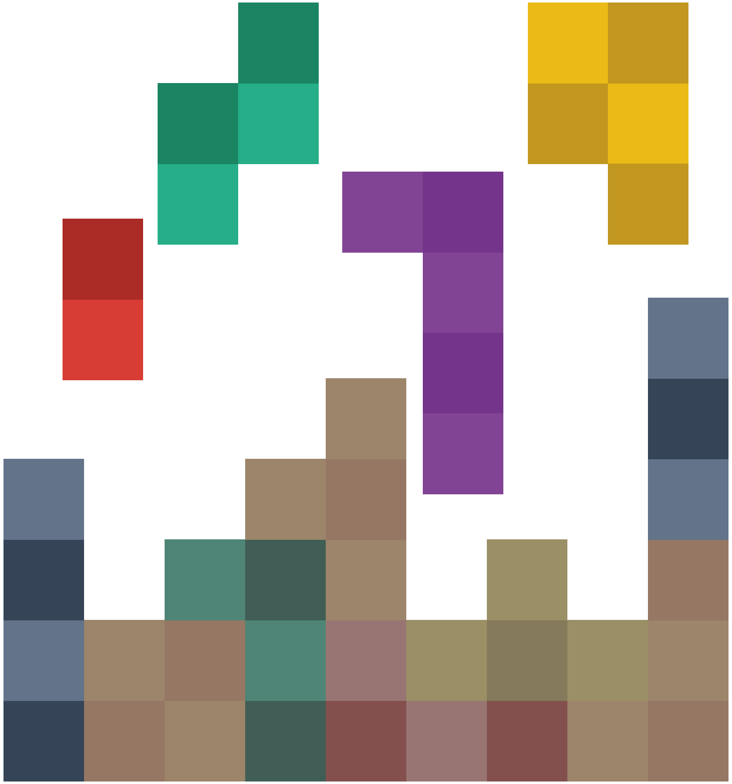
Functions do not necessarily make code shorter (at first)! Compare:

```
indexATG = [n for n,i in enumerate(myList) if i == 'ATG']
indexAAG = [n for n,i in enumerate(myList) if i == 'AAG']
```

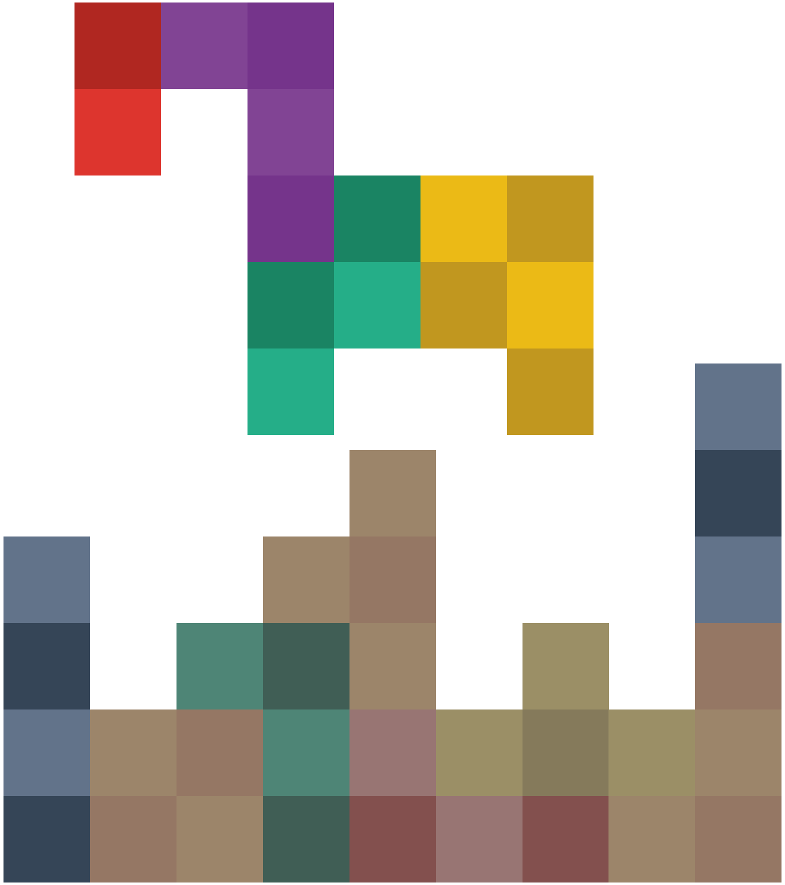
```
def indexString(inputList,z):
    zIndex = [n for n,i in enumerate(li) if i == z]
    return zIndex

indexATG = indexString(myList,'ATG')
indexAAG = indexString(myList,'AAG')
```

Think in building blocks!

Small, cohesive units are much better than…



… a customized behemoth!



Your turn: visualize your code!

Choose:

- Make a screenshot, process it in paint, powerpoint, or your favorite editor;
- Copy paste your code to a text editor, and use markers.

The objective is for you to 'see' your code!

- Yellow denotes scripted, unstructured code *(basic, sequential lines of instructions)*
- Purple denotes functions or other structured code *(e.g. for-loops, conditionals, etc.)*
- Green denotes comments (or comment blocks) *(consider combining this with yellow for heavily commented code)*

Again, make notes in your code (`#TODO`!) if you see:

- **Scripted code**: this can be a function
- **Structured code**: this should be re-structured

What can you learn from your colleagues today?

Your turn: make a function

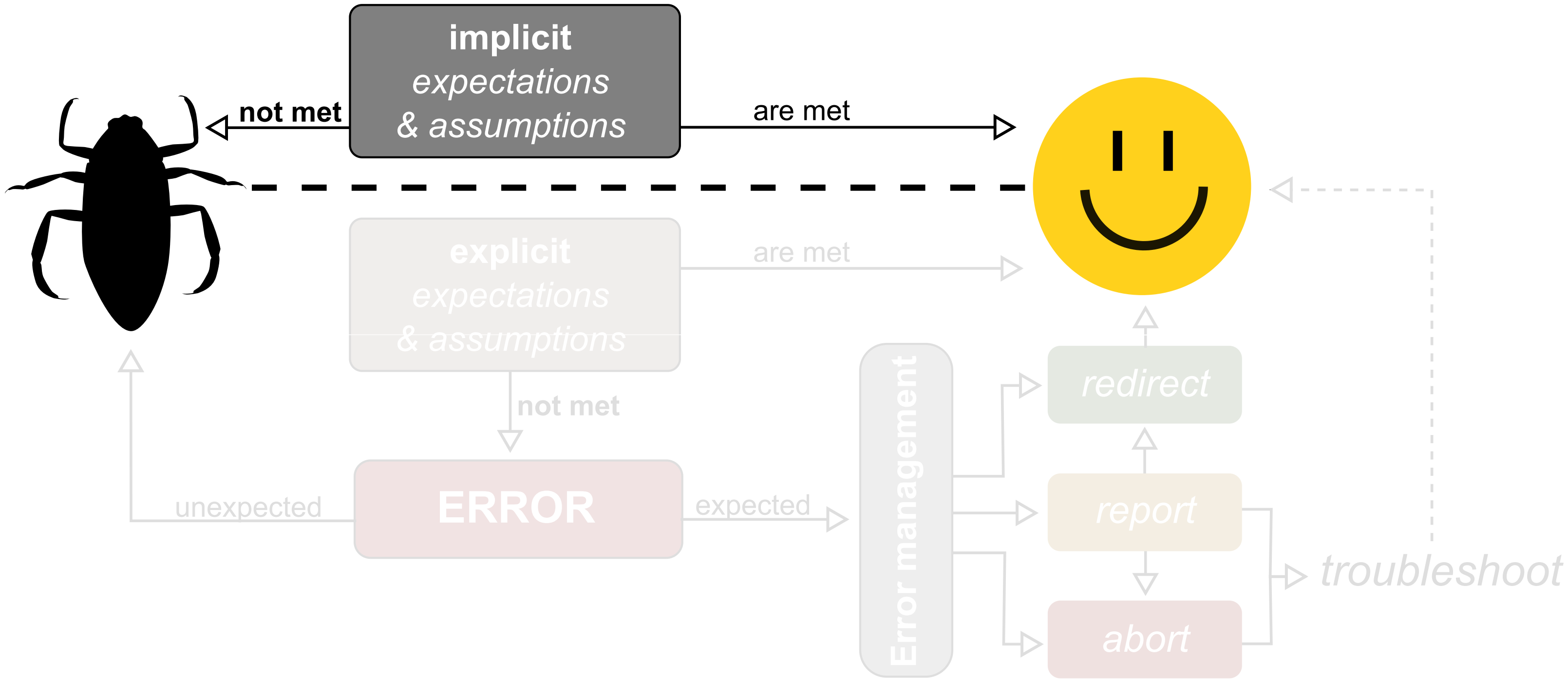
You have visualized your code. Use your findings to improve it!

- **Preferably**: take scripted code and turn it into a function, *or* split an existing function into two or more functions.
- If there is no function to work on: try and address the readability of your code.

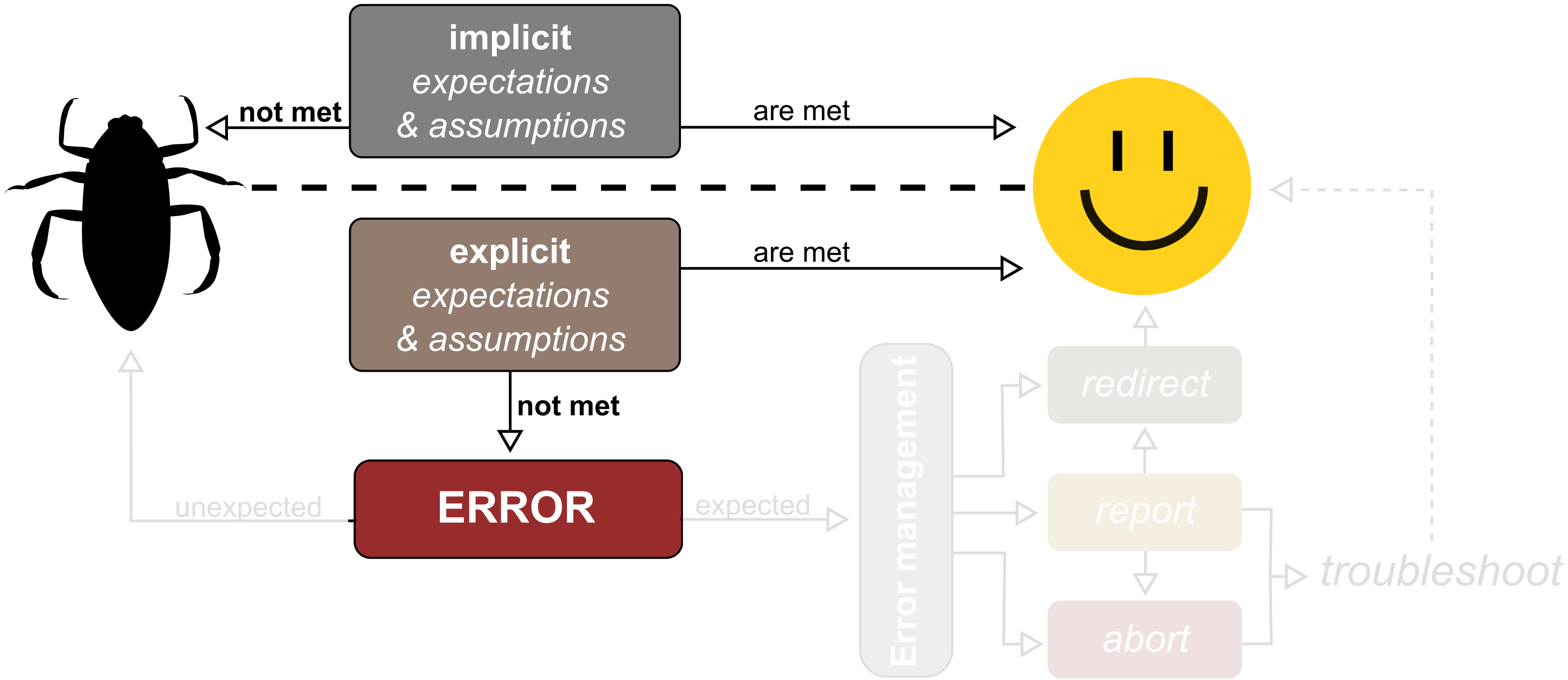
However: for future exercises you will need at least one function, preferably with parameters, in your code! For example:

```python
def my_function(param_a, param_b):
  if param_b == 99:
    return None

  if param_a == 100:
    do_something(param_a)
  else:
    do_something_else(param_a)
```

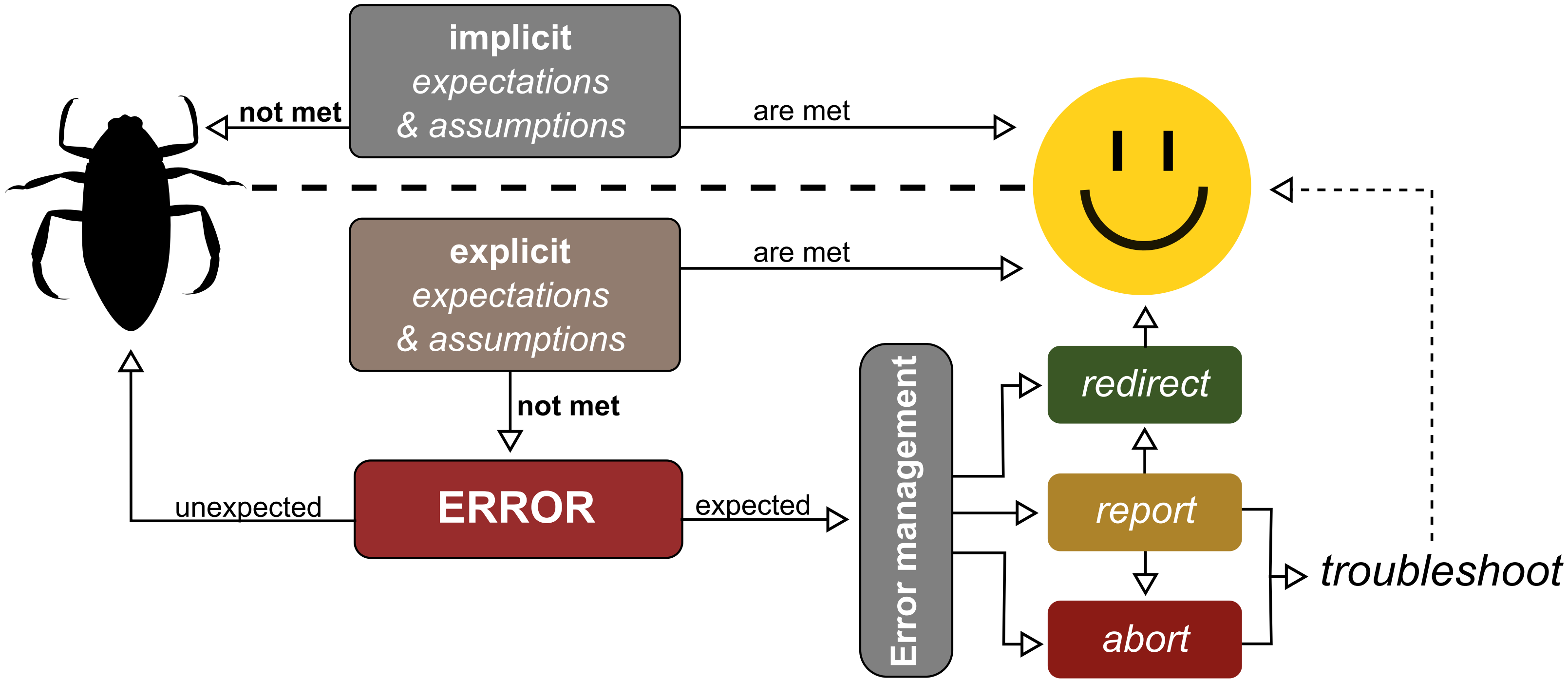
Code robustness



Code robustness



Code robustness



Error management

Protect the user:

- Make assumptions and expectations explicit.
    - check values before processing them
    - identify and manage exceptions
- Produce errors when expectations are not met.
- Consider error options, and perform error management:
    - redirect the program
    - log or report the error, to allow the user (or developer) to troubleshoot
    - if necessary: abort the run

Advanced robustness: unit tests

Protect the developer (you!)

- Test the expected behavior of your functions:
    - Confirm a known output given a known input
    - Do errors get produced as expected when the input calls for it?
- Capture unexpected errors to identify further options for error management
- You can automate running tests when pushing to Github using Continuous Integration
- Tests are **definitely** worth learning when your project increases in size!

More on tests later...

Throwing an error

```
def read_vector_value(index=0, my_vector=[10,5,4,12,25]):
    if index > len(my_vector) - 1:
        raise IndexError('Index higher than vector length.')
    return my_vector[index]

read_vector_value(index=6)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): IndexError: Index higher than vector leng
```

Why not simply adjust the function output?

```
def read_vector_value(index=0, my_vector=[10,5,4,12,25]):
    if index > len(my_vector) - 1:
        return None
    return my_vector[index]

print(read_vector_value(index=6))
```

```
## None
```

Because it is unclear if `None` is expected behavior or indicative of a problem.

Warning message without breaking

An error breaks code execution

```
read_vector_value <- function(index=1,my_vector=c(10,5,4,12,25)){
  if(index>length(my_vector)){
    stop("Index higher than vector length.")}
return(my_vector[index])
}

print(read_vector_value(index=6))
```

```
## Error in read_vector_value(index = 6): Index higher than vector length.
```

Capture the error but release a warning

```
read_vector_value <- function(index=1,my_vector=c(10,5,4,12,25)){
  if(index>length(my_vector)){
    warning("Index higher than vector length.")
    return(NA)}
return(my_vector[index])
}

print(read_vector_value(index=6))
```

```
## Warning in read_vector_value(index = 6): Index higher than
## vector length.
```

Redirecting with exceptions

If you do not want to interrupt your script when an error is raised: use try/catch ('except' in Python). NB: Note that Python allows you to distinguish by error type!

```
try:
  read_vector_value(6)
except IndexError:
  print("This is an exception")
```

```
## This is an exception
```

```
try:
  read_vector_value(6)
except ArithmeticError:
  print("This is an exception")
```

```
## Error in py_call_impl(callable, dots$args, dots$
```

In R you can use tryCatch():

```
tryCatch({
    read_vector_value(6)
}, error = function(e) {
    print("This is an exception.")
})
```

```
## [1] "This is an exception."
```

Validating input

Consider early statements in the script to validate (data) input.

With if/else:

```
if not protein_data:
  raise ValueError("Dataset cannot be empty")
```

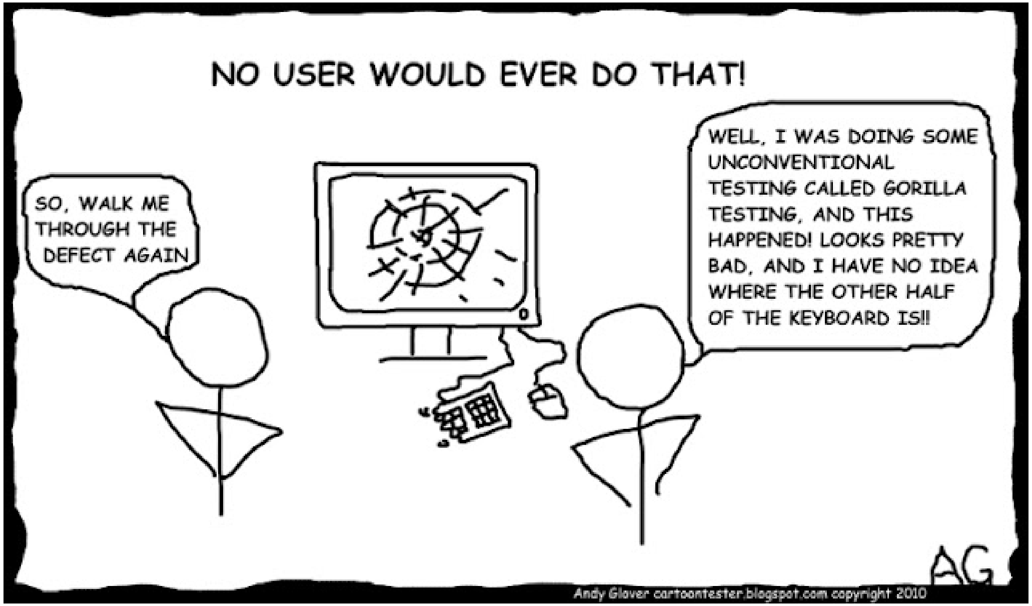
With try/catch:

```
tryCatch({
  do_something_that_might_go_wrong(protein_data)
}, error = function(e){
  log(e)
}, finally = {
  cleanup(protein_data)
})
```

Expectations and assumptions

Expect the worst

- use of wrong input values for functions
- malformed text input
- wrong data types


NO USER WOULD EVER DO THAT!
SO, WALK ME THROUGH THE DEFECT AGAIN
WELL, I WAS DOING SOME UNCONVENTIONAL TESTING CALLED GORILLA TESTING, AND THIS HAPPENED! LOOKS PRETTY BAD, AND I HAVE NO IDEA WHERE THE OTHER HALF OF THE KEYBOARD IS!!
AG
Andy Glover cartoontester.blogspot.com copyright 2010

Source: cartoontester

Your turn: explicit expectations

Identify assumptions in your code

- What assumptions/expectations exist on your data or (user) input?
- What assumptions/expectations exist on the input of (a) function(s)?

Make the input/data assumptions explicit

- **Option 1**: Explicitly state assumptions on data or input in your README.md.
- **Option 2**: Write a piece of code that tests the validity of data/input, and reports an error if the expectations are not met.

Test the input for a function

- Modify the code inside your function to
  - check the value of the arguments passed to your function using if/else statements;
  - raise an error in case an argument is out of the range of acceptable values.

Unit testing

Unit testing is a generic testing approach.

Your software is tested by focusing on smaller units, for instance a series of functions or class.

Extra packages\imports are needed

- in R with the testthat/testthis packages
    - https://github.com/r-lib/testthat, https://github.com/r-lib/testthis
- in python with pytest, unittest
    - https://docs.python.org/3/library/unittest.html

Running unit tests

Code editors/IDEs such as visual studio code, RStudio, Pycharm…

- Integrate functionalities to run and show the results of unit tests
  - E.g., RStudio in the build menu -> test package

1. Create a unittest file

   ```
   usethis::use_test("hello")
   ```

2. Edit the file test-hello.R in the tests folder (created by usethis)

   ```
   test_that("multiplication works", {
   expect_equal(2 * 2, 4)
   })
   ```

3. Run tests

   - In RStudio using the menu **or**

     ```
     devtools::test()
     ```

Example

Testing our read_vector function

```python
import unittest
class TestStringMethods(unittest.TestCase):
    def test_retrieval(self):
        self.assertEqual(read_vector_value(0), 10)

    def test_error(self):
        with self.assertRaises(IndexError):
            read_vector_value(5)
```

Run the tests by calling `unittest.main()`

Or, when working in a notebook:

```python
unittest.main(argv=['trick to make it work in a notebook'], exit=False)
```

```
## <unittest.main.TestProgram object at 0x0000018EF1D00A30>
##
## ..
## ----------------------------------------------------------------------
## Ran 2 tests in 0.001s
##
## OK
```

Making a test fail

```python
import unittest
class TestStringMethods(unittest.TestCase):
    def test_retrieval(self):
        self.assertEqual(read_vector_value(0), 10)

    def test_error(self):
        with self.assertRaises(IndexError):
            read_vector_value(5)

    def test_retrieval_wrong(self):
        self.assertEqual(read_vector_value(0), 11)
```

Result of faulty test

```
## <unittest.main.TestProgram object at 0x0000018EF1C8FA60>
##
## ..F
## ======================================================================
## FAIL: test_retrieval_wrong (__main__.TestStringMethods)
## ----------------------------------------------------------------------
## Traceback (most recent call last):
##   File "<string>", line 10, in test_retrieval_wrong
## AssertionError: 10 != 11
##
## ----------------------------------------------------------------------
## Ran 3 tests in 0.000s
##
## FAILED (failures=1)
```