Software developing paradigms

WRITE YOUR CODE LIKE A SOFTWARE DEVELOPER

Scientist vs Software Developer

- What's the difference between a "typical" scientist and a software developer?

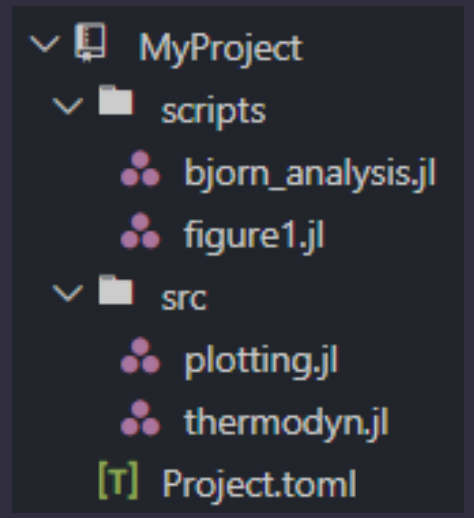
1. Writes code for themselves
2. Copy/pastes 100 loc to change 1…
3. No lectures on writing good code
4. Not very good at optimizing code
5. Good at inventing algorithms

1. Writes code for other people
2. Writes minimal and extendable code
3. Had lectures on writing good code
4. Knows how to optimize code
5. Good at designing APIs
6. Reads r/ProgrammerHumor

you want these skills!

Separate low-level and high-level

- Mentally separate your code into low-level and high-level parts: ***source vs. scripts***
- Low-level = source: hardcode computations, generation of data, optimized code, definition of functions/structures used throughout the codebase, plotting recipes, typical data analysis pipelines, …
  - Each function in low-level is of general use and reused
  - Low-level should never have code duplication
- High-level = scripts: running/submitting simulations, listing/loading results, visualization, analysis applied to specific data, creating notebooks for presentations…
  - ***High-level should never define new functions/methods***



- The user of the code should only see the low-level parts when:
  - Adding a new low-level functionality OR fixing a bug. ***NEVER for copy pasting!!!***
  - `src` must be ***crystal clean***, so clean your mama would be proud for seeing it

Example of “source vs scripts”

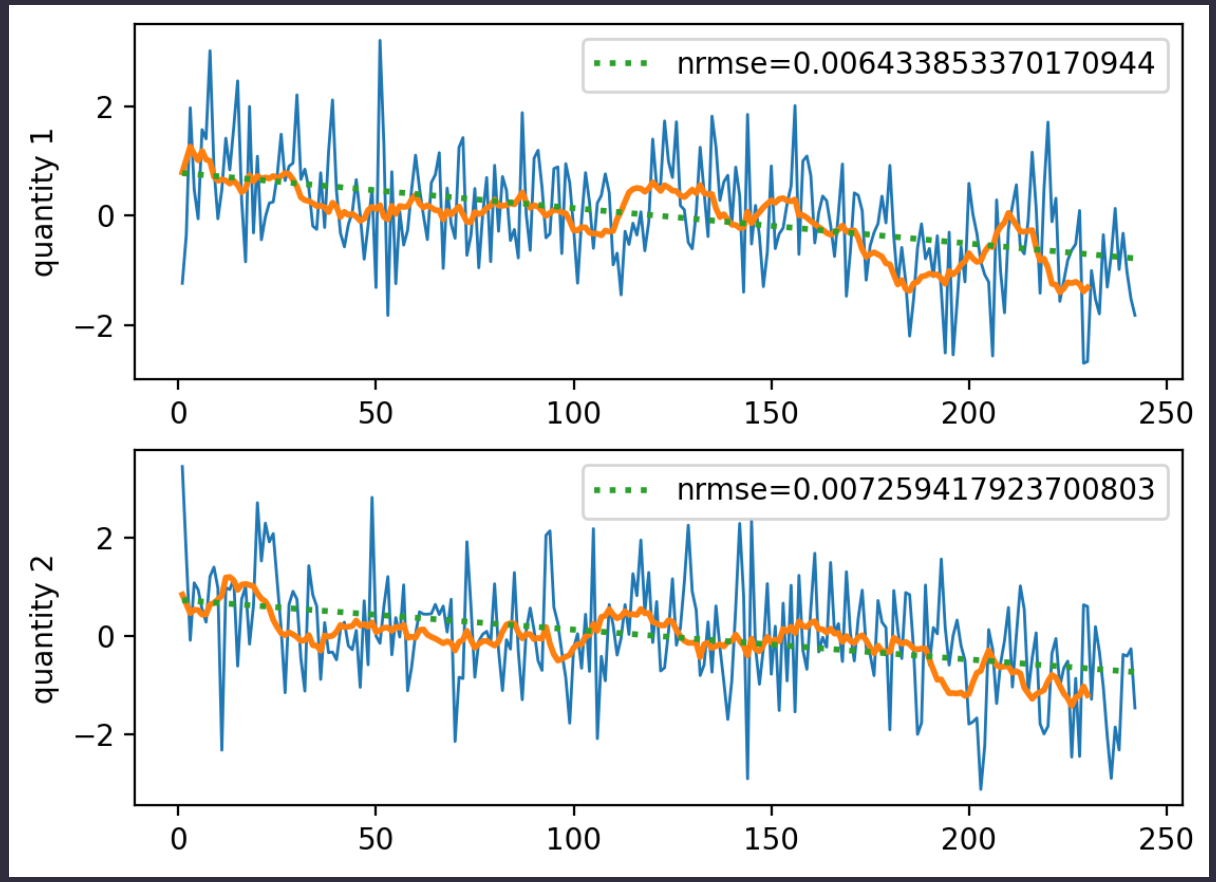
- In case you’re thinking “*this isn’t possible in a “real” science project…*”
- Here’s how I applied these guidelines in ~~my latest project~~ three years old project
  - Honestly, good test if I actually wrote good code…

- The project is called “VelocityMTD”,
  - Correlations between timing and intensity in music performances

- src: correlations.jl, vel_normalization.jl
- script: drums_scatter_single.jl
  - Notice: scripts only use functions from `src`
  - Notice: scripts have only 1 level of abstraction

Exercise: monolithic script

- Get the `plot_timeseries_monolithic` file, which is a script that produces this plot:



- Understand its building blocks
- Separate the script into two files:
  - `_script` and `_src`.
- In `src`, make the building blocks into reusable functions
- In `script`, include the source file, and use the defined functions
- Your `script` should not be more than 20 l.o.c.

Design a high level API for `src` and use it!

- To use your source code, you need an Application Programming Interface

- Design an API for your code first on paper/mentally
  - *think how you would like to use your code and obtain results*
  - *think ahead: what extensions do you expect?*
  - *did you design your API to be general enough?*

wear the hat of the user

- Then write this API on code
  - *Is it minimal? Is it extendable? Is it modular?*
  - *Is it understandable and simple to reason for logically?*
  - *Does it have arbitrary restrictions that it should not have?*
  - *Is it too specific to your setting, even though it doesn't have to?*

wear the hat of the developer

Remember: you're both unfortunately!

- Pro tip: try it so that there is ***only one way*** to do something; similar ways to do something should do some setup, and then call the one way to the thing.

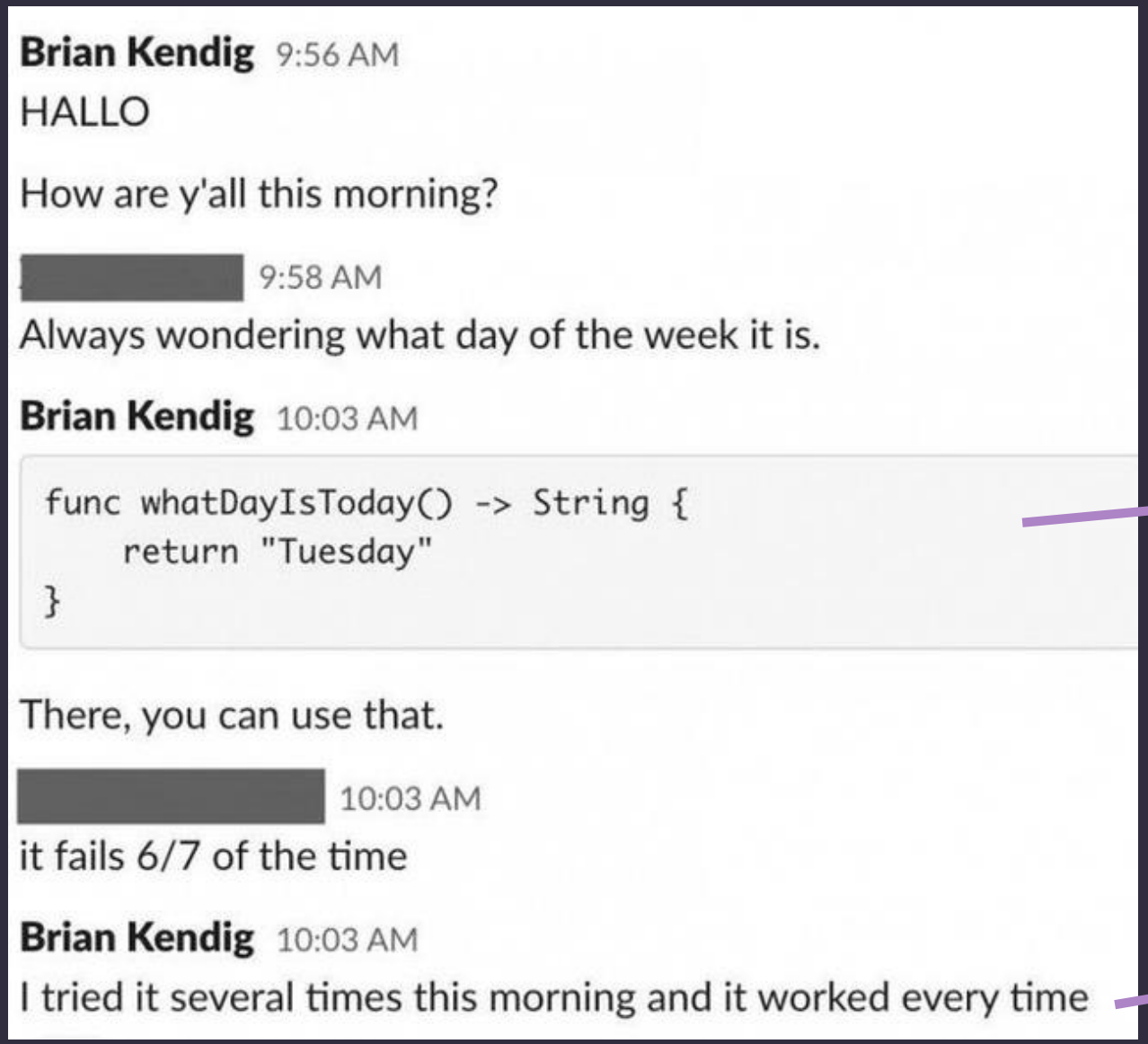
Exercise: Design an API for monthly means

- Process description: aggregation of a vector `x` with a time vector `t` into monthly means. The time vector `t` has values that are proper `Date` instances.
  - To be specific: for all dates in first month the values of `x` are averaged, then repeat for next month…
  - The function then returns `m,y` with `y` the means and `m` the new monthly time vector
  - If you are not familiar with handling Date types, do the next exercise with running means
- Let's now write the code for it. Ensure it works by testing it with a simple case of daily timeseries where the value at any date is the month number of the date
- ***Part 1, 10 minutes, then: Part 2: 20 minutes (yes, amount of time is way too short)***
- Part 2: What is the most reasonable extension to this codebase we can expect?
  - Yearly averages, daily averages, weekly averages…
  - Standard deviations instead of averages, or other moments, or user-specified functions
- Is it straightforward to apply this extension to the current code base?
- What modifications can we do to indeed make this extension possible?
- Test it by calculating "summer/winter" averages (summer = months ∈ (3,4,5,6,7,8))

Exercise: Design an API for running means

- Process description: running means of a given vector, first with a “rectangular window” version, i.e., n-point averages.
- What configuration options should the user have…?
- Let’s now write the code for it

- What is the most reasonable extension to this codebase we can expect?
  - Different kind of averaging “windows” such as Gaussian or many other
  - Arbitrary user-specified selection of points for the average, e.g., every 2nd point, etc.
  - Standard deviations instead of averages, or other moments
- Is it straightforward to apply this extension to the current code base?
- What modifications can we do to indeed make this extension possible?



some sloppy code that did a specific thing that you wanted at a specific context and time of day

the lack of a context-agnostic viewpoint forbode you from seeing the problems with the code

Benefit of having `src`: Unit tests

- Scripts are by nature one time things. But `src` is re-usable…
- How do you ensure that your `src` code (a) does what it's supposed to, and (b) continues to work as time progresses…?

- You write ***unit tests!***
  - Small pieces of code that are run and versus an expected outcome
  - Help you identify potential bugs or inaccuracies because of testing in context-agnostic situations and/or more situations than your single specific use case
  - Give confidence that your scientific results are numerically valid
  - Can be automated (see later slides on continuous integration in [block4])
  - Ensure that everything that used to work, ***still works***

- Test-driven-development is the discipline where one first writes the tests for some functionality, and then its implementation. Try it out!

Actually writing a test suite

Julia

```julia
using Test # module from Standard Library

@testset "MyPackageTests" begin
    @testset "arithmetic" begin
        include("math_tests.jl")
    end
    @testset "trigonometric" begin
        include("trig_tests.jl")
    end
end

# e.g., "math_tests.jl" has:
using Test
@test 1 + 1 == 2
@test 1 - 1 == 2
```

```
Test Summary:    | Pass  Fail  Total
MyPackageTests   |    3     1      4
  arithmetic     |    1     1      2
  trigonometric  |    2            2
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

Python

```python
# content of test_math.py
def test_math():
    assert 1 + 1 == 2
    assert 1 - 1 == 2

# Run it with:
pytest test_math.py
```

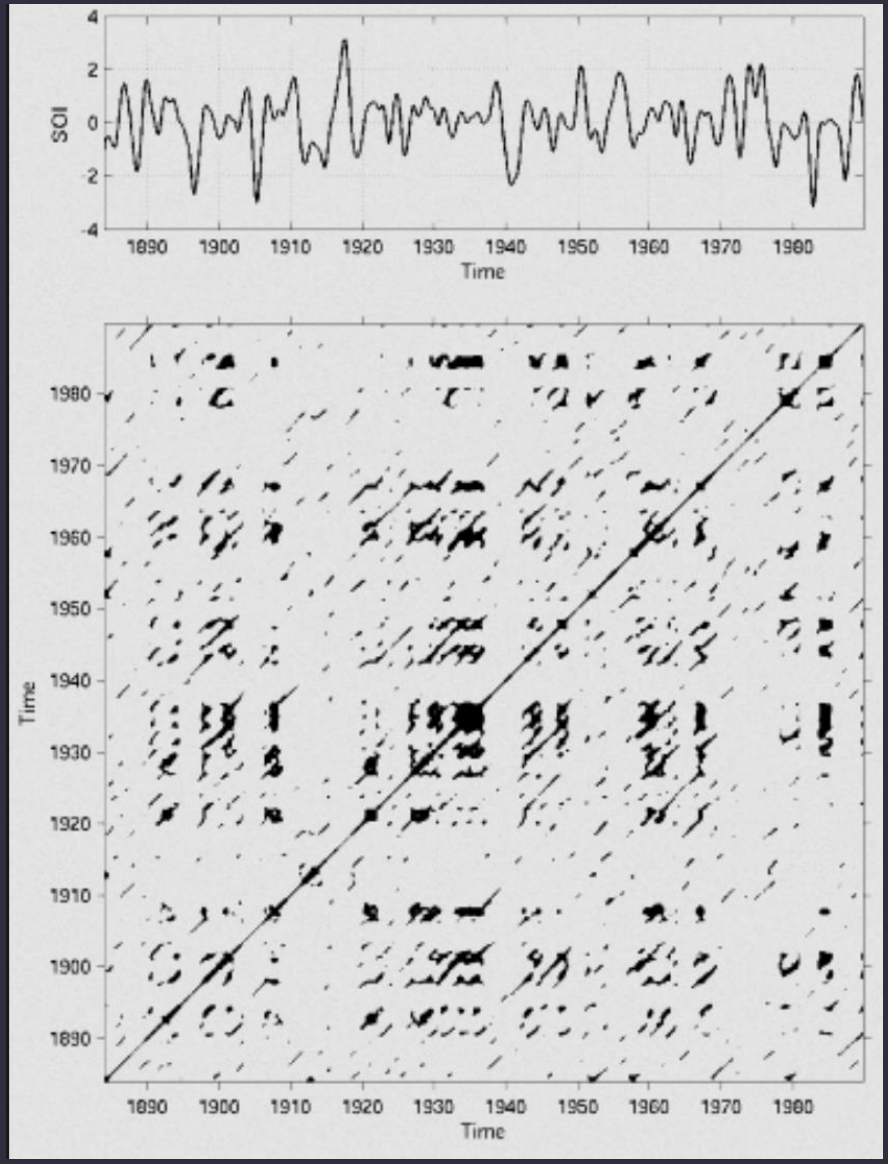
```
============================= test session starts ==============================
platform darwin -- Python 3.9.13, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/lkluft/Desktop/example-python
plugins: anyio-3.6.1
collected 1 item

test_math.py F                                                           [100%]

=================================== FAILURES ===================================
_________________________________ test_math _________________________________

    def test_math():
        assert 1 + 1 == 2
>       assert 1 - 1 == 2
E       assert (1 - 1) == 2

test_math.py:6: AssertionError
=========================== short test summary info ============================
FAILED test_math.py::test_math - assert (1 - 1) == 2
============================== 1 failed in 0.06s ===============================
example-python (main %)
```

Properties of good unit tests

- ***Actually unit:*** test atomic, self-contained functions. Each test must test only one thing, the unit. When a test fails, it should pinpoint the location of the problem.
  - Testing entire processing pipelines (a.k.a. integration tests) should be done only after units are covered, and only if resources/time allow for it!
- ***Known output / Deterministic:*** tests defined through minimal examples that their result is known analytically are the best tests you can have!
  - If random number generation is necessary, either test valid output range, or use seed for RNG
- ***Robust:*** Test that the expected outcome is met, not the implementation details
  - Test that the target functionality is met without utilizing knowledge about the internals
- ***High coverage:*** the more functionality of the code is tested, the better
- ***Clean slate:*** each test file should be runnable by itself, and not rely on previous test files
- ***Fast:*** use the minimal amount of computations to test what is necessary
- ***Regression:*** Whenever a bug is fixed, a test is added for this case
- ***Input variety:*** attempt to cover a wide gambit of input types

An example of good unit tests

- Recurrence plot: method to transform a timeseries into a Matrix encoding recurrences
- Further analysis quantifies the structures in the plot: diagonal lines, vertical lines, distances, etc…
- ***BAD test***: create recurrence plot from a saved timeseries, calculate the recurrence metrics using your code, and then write tests that confirm code gives metrics...
  - You only test that the interface doesn't break over time.
  - You don't actually test the implementation of the science!
- ***GREAT test:*** write down a small 9x9 matrix with recurrences at specified positions, and analytically / by hand calculate all measures. Then test that the code gives the expected results!
  - github.com/JuliaDynamics/RecurrenceAnalysis.jl → /test/smallmatrix.jl
- Image: recurrence plot of ENSO, from Wikipedia



Brenan Keller
@brenankeller

A QA engineer walks into a bar.
Orders a beer. Orders 0 beers.
Orders 99999999999 beers.
Orders a lizard. Orders -1 beers.
Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM · 30 Nov 18

Exercise: Write unit tests!

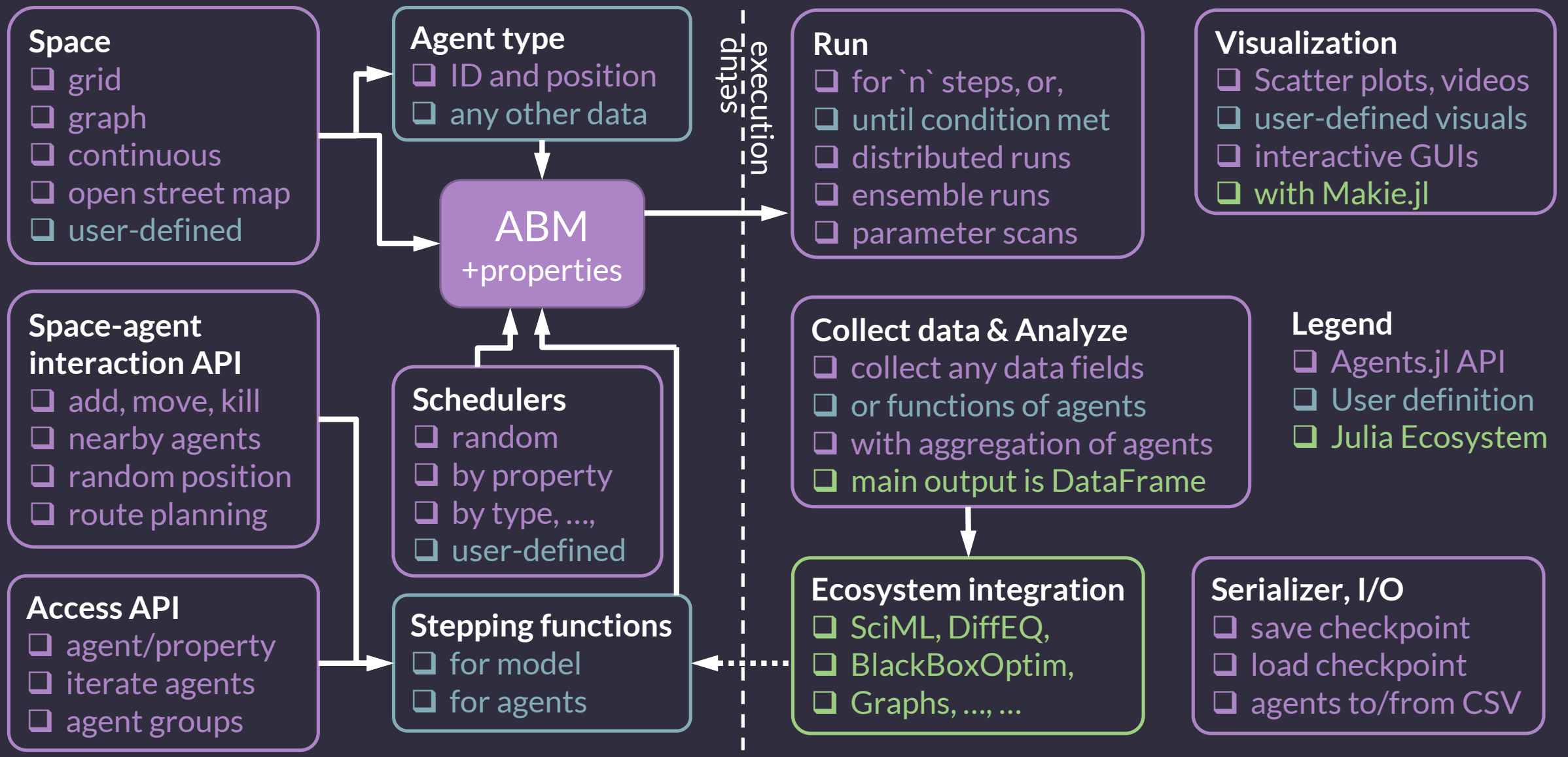
- Use the function of the previous exercise with monthly means or running means
- What should be the meaningful unit tests you should write for it?
- Write them and run them!
- Are they “good”?
  - (unit, clean slate, deterministic, input variety, high coverage, fast, robust)

- ***Maximum time: 10 minutes***

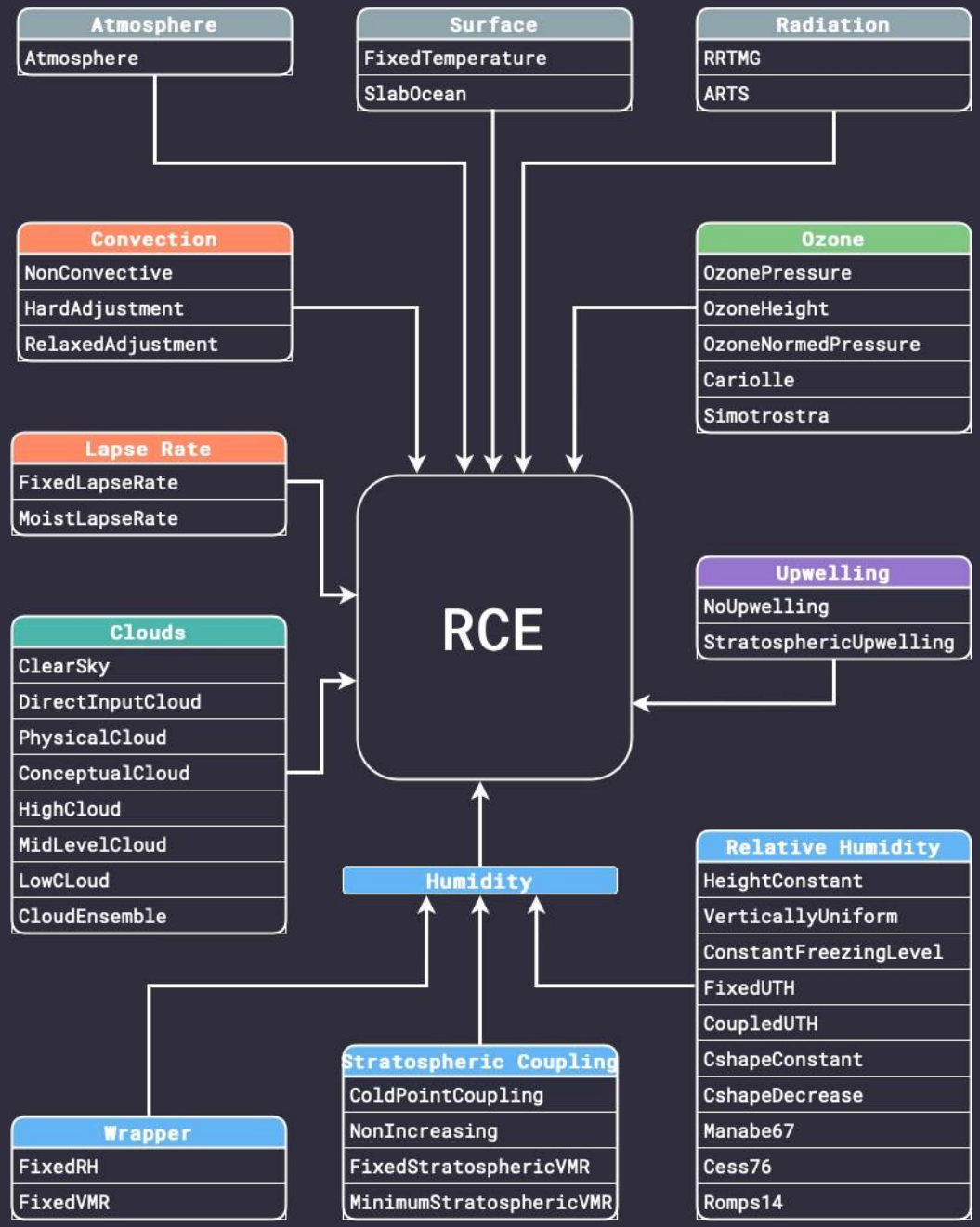
Modular software

- One of the most important aspects of modern software design
- A ***modular software*** is
  - Composed by individual and smaller components that combine together to yield the whole
  - Each of the components is self contained: it has a purpose on its own, can be instantiated on its own, and if sensible, ***can even run on its own***
  - The way the components are defined, and the way they combine together, is declared explicitly by well-thought-out ***interfaces*** (see next slides)
  - From a user standpoint, the components must be instantiable, and combine together, as effortlessly as when describing them in human language. So it must really take only a few of lines of code to use a modular software, and the bulk of code must be only in the instantiation part (because it typically has many configuration options the user wants to decide on).
- A modular software helps a user by ***allowing them to focus on the composition*** (i.e., the science, i.e., putting things together), and not on making the composition work!
- It is also the most robust way for reproducible, and hard-to-break, software

Example of modular code: Agents.jl



Another example: konrad



Modular design is best executed by Interfaces

- An “interface” is a set of functions that types/classes must implement to be part of said interface. This set of functions logically gives the…
- …functionality that naturally makes the objects part of the interface
- ...functionality that allows the objects to modularly connect to and be used by a larger ecosystem. I.e., the Input/Output of the interface.
- One of the best parts of modular software is that it “forces” you to design good interfaces

- Function-based interfaces are better than class-based, because:
  1. You can define new interface functions after creation of a type/class
  2. You can define fully independent classes/types that adhere to same interface (and hence no reason to inherit all information/methods from a parent)
  3. You don’t have to modify the source code of existing packages to add new methods
  4. Strongly recommended: The Unreasonable Effectiveness of Multiple Dispatch

My slice of life here

- By night I live as software developer; has this affected my science career…?
- You bet! I have understood that…

documenting my code well is extremely important for my future self

designing great APIs makes me and my co-workers work faster and extend easier

complicated code with many functionalities can still be written in 100 lines (smartly)

people want to use my code, and they want to contribute back

making the code base a proper package makes me secure about my own projects

documenting my code results in me knowing the potential of my code much better

But dude, how do I get these skills???

- practiceeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
- Writing code is a ***craft*, *an art***. You don't learn music by buying CDs, you don't learn painting by buying paintings, and similarly, you don't learn coding by only using existing libraries!!!
- Write as much source as possible, and actively improve it
- Write your own versions of library functions for practice!
- ***Contribute to open source: you get mentoring in exchange for contributions***
  - ***By far best way to improve.***

Your code → professionally developed

- Does your code operate based on the "source and scripts" principle?
- Make it so by implementing a well thought out API for source!
  - Split your code into source and script files
  - Source files are composed of small functions
  - Scripts only call functions from source, and therefore have only ***one level of abstraction*** (for-loops still count as one level)

- Write a test suite for your source code. Make it "good"!
- Are your tests biased to your exact scientific problem? That's bad!
- Run the test suite! Do all tests pass?
- At the end, add another tag to your code called "professional" (ha ha ha)

- ***I can't stress this enough: write tests for your source code! You will be surprised how often you f***ed up and believed you wrote something that was correct, but actually wasn't!***

Package development

- ***Consider:*** publish `src` as a package of your programming language, that your scripts use, or contribute your `src` functions to an existing package they fit at!
- Package development essentially boils down to:
  1. Having a well defined API
  2. Have good, extendable source code that implements it
  3. Maybe performant code (depends on application)
  4. A trustworthy test suite
  5. Documentation! That's what we haven't covered yet, but it's coming! in block 5!
- *"...But all of these must need an entire team of software developers, I'm just a poor boy from a poor family!"* you might say. Nay! With modern software and modern tools, these become much easier than you'd think!
  - Julia: 84% of published packages have at least 20 lines of tests, 88% have at least a README or /docs, yet the median number of contributors is just 2... It can be done!
  - (see course by Tim Holy)