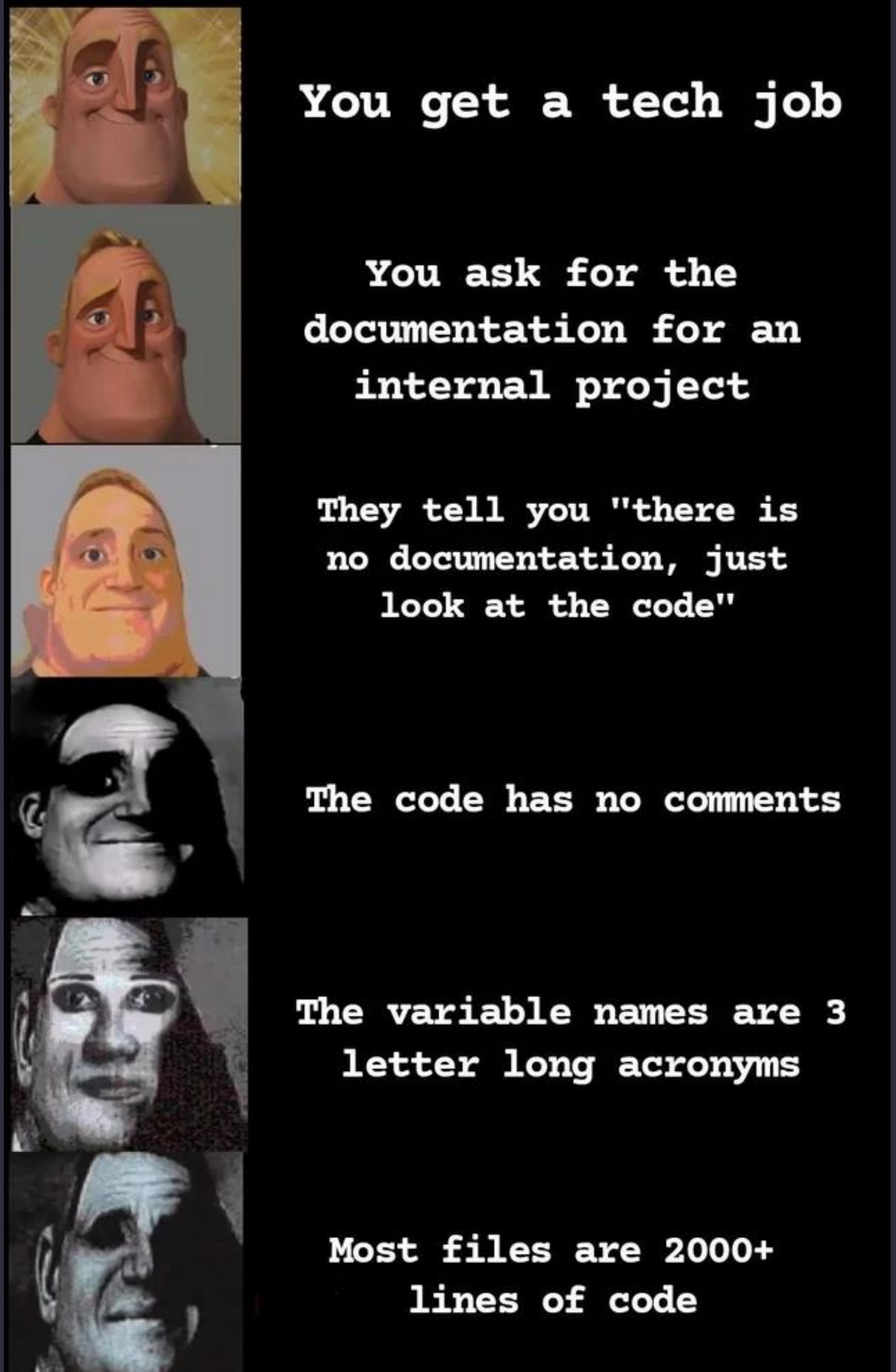
Documenting software

WRITING AND MAINTAINING AN EXCEPTIONAL DOCUMENTATION THAT CONVEYS INFORMATION EFFICIENTLY AND INTUITIVELY

I don’t need the documentation, I can make it work!

Why even bother with documentation...?

- Because you want people to use your code! But...
  - If people can't figure out what is the purpose of your code, how to install your code, or how to use your code, they won't use it! Sounds obvious, right? ***Because it is.***
- Learning a software by its source code is tremendously inefficient (if possible!)
  - *"There is a **vanishingly small** number of people who will source-dive to use any code out there, compared to the people who will use your code when properly documented."*
    ~ Eric Holscher, co-founder of readthedocs and writethedocs
- Good documentation, like good code, will motivate contributions!
  - You can only get contributions after having users, which happens after having a documentation
- ***Documentation is the most important part of your software if you want >1 user count!***


You get a tech job
You ask for the documentation for an internal project
They tell you "there is no documentation, just look at the code"
The code has no comments
The variable names are 3 letter long acronyms
Most files are 2000+ lines of code

sounds familiar…?

Let’s make a *better world!*

Overview of this block on Documentation

- What should be in the docs, what's most important
- How to write good quality documentation content
- Documentation strings
- The overarching/introductory tutorial
- Tips & tricks (T&T) for better clarity and maintenance

Julia agnostic! General guidelines!

- Documenter.jl+Literate.jl for generating runnable tutorials and examples
- Documenter.jl for composing documentation pages
- Documenter.jl for deploying documentation online on GitHub automatically
- Other goodies in JuliaDocs as time permits

Julia specific!

Setting expectations right!

- This workshop total time: ***~3hours***
- Real documentation time investment: ***~2weeks – 6months fulltime job!!!***
- it is unrealistic to have a fully-fledged documentation after this workshop!

- ***Actual Workshop Goal:***

1. have a solid understanding of the qualities underlying a good documentation
2. have a solid foundation of the documentation of your own codebase
3. have a solid grasp on the technical skills required to create a documentation

Documentation

- What's the role of documentation?

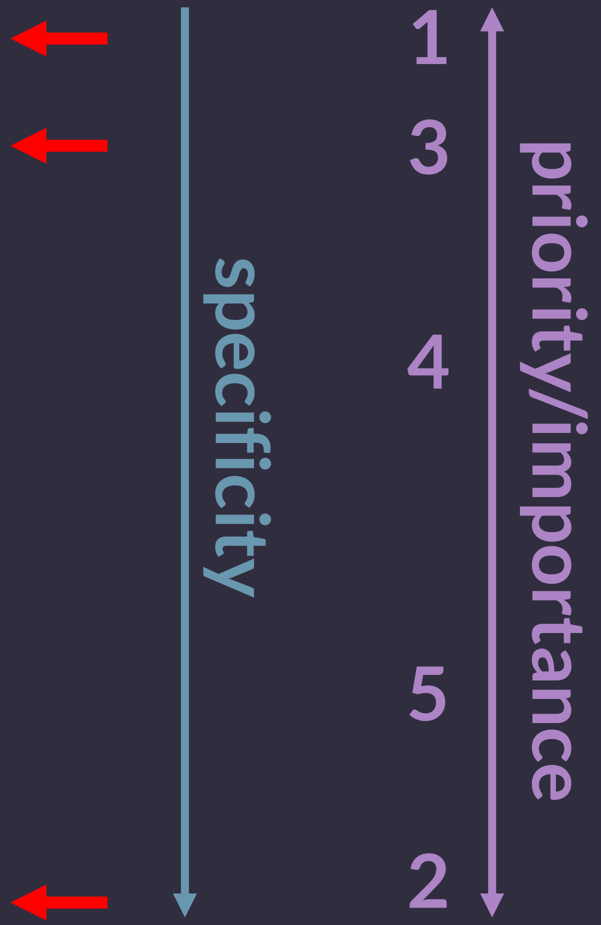
1. Teach new users the software
2. Serve as reference for existing users
3. Define the public API (how to use the software) → also defines what is a bug or not!
   (remember: only the ***intended*** behavior of software can be described!)

- A ***good documentation*** will make the user:
  - Understand what this software is about
  - Get an immediate overview of the features of the software
  - Find the information they need fast
  - Find it where they expect it to be
  - Understand the information quickly
  - Have no ambiguity after reading the docs (no guessing!!!)

Documentation components

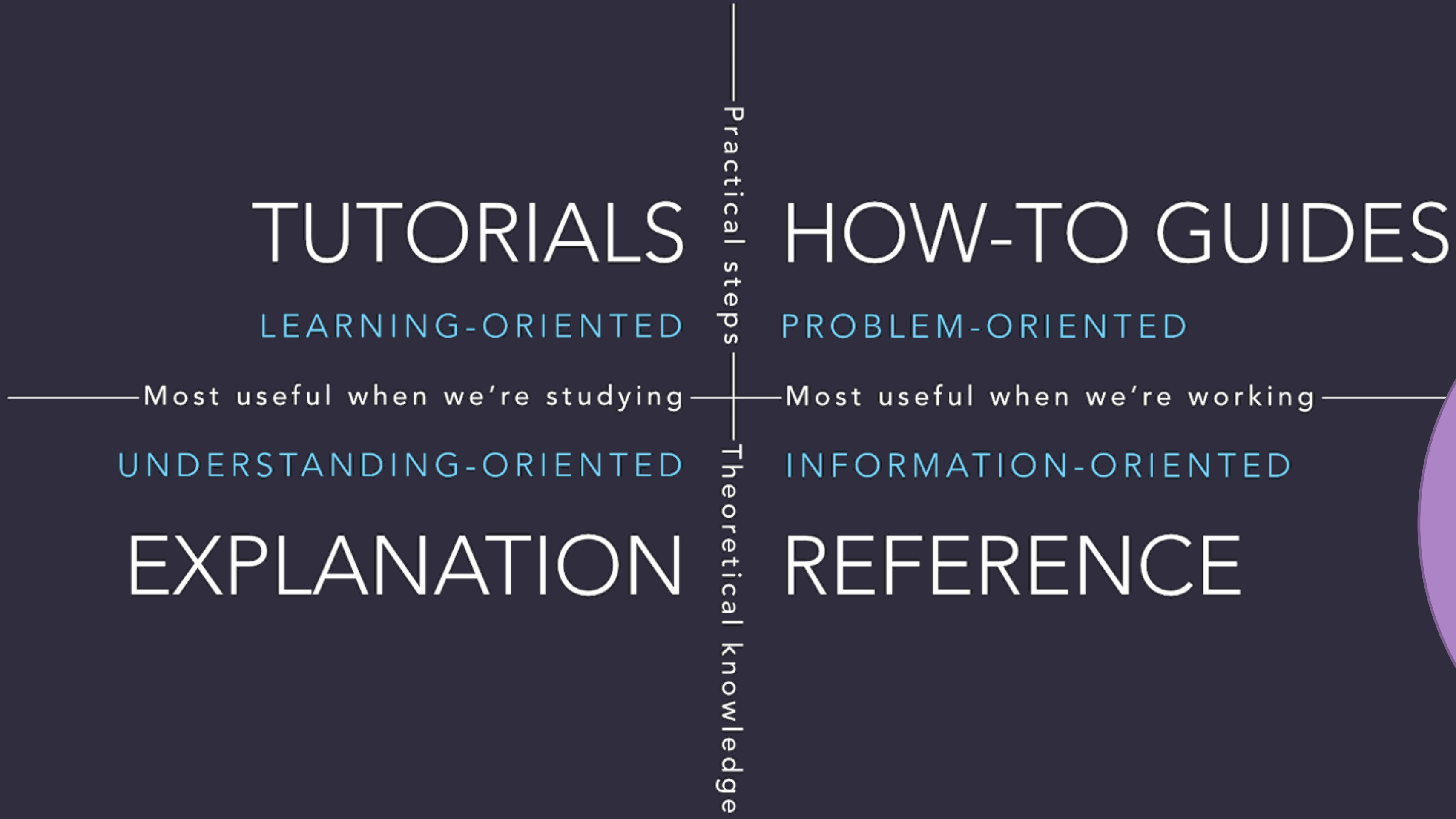
Several levels of depth of exposition framework:

- **Highest level**: Summary of what the software does
- **High level**: Overarching tutorial on the core of the software
- **Intermediate level**: Examples
  - Showcase end-to-end usage of specific or advanced functionality
  - ***Examples must be RUN. Not written! More on that later!***
- **Low level**: internals and explanations
- **Lowest level**: Public API / documentation strings
  - *(Good Scientific Code Workshop teaches functional code)*



Documentation components: alternative

- The Diátaxis system (from https://diataxis.fr), it is an alternative popular to structuring the components of the documentation



I prefer the "**depth of exposition**" approach over Diátaxis: 1) Diátaxis doesn't attribute order of importance, 2) there isn't a difference between "tutorials" and "how-tos" in my experience

Software summary

Software summary = README

- Summary should be in the thing people see first: the README file
- Should be a couple paragraphs long and should say:

- what the software does
- what are its major features
- what makes it unique
- how it compares to alternatives
- links to docs, citation, CI, authors…
- links to related software

- Here are two example README files: Agents.jl, OnlineStats.jl
- Personally: against having any code documentation/example in the README, unless the README is the entirety of the documentation

T&T: Julia automatically makes the README.md the documentation string of a Package (in standard folder layout src/PackageName.jl)!

The Markdown syntax

- We will be writing documentation using Markdown, so we better learn the basics!

~~~markdown
# Markdown intro
Markdown is a way to style text by using plain text that is later rendered into HTML.

## Basics
Text prefaced with one to six `#` and then an empty space creates headers.
- Markdown can do lists like the one you read now with `-`
- Make text **bold** or _italic_, and can even do [website links](www.google.com)
- Markdown can also do math but the syntax depends on the renderer.

1. Markdown can also do numbered lists, like the one you read now.
2. Tables, in-line `code` (backticks), code blocks (three backticks), and many more!
3. See https://www.markdownguide.org/ for more syntax guidance!

A code block with Julia highlighting (identing by 4 spaces also makes code block)
```julia
rand()
x = 1 + 1
```
~~~

Note: there is no universal Markdown ☹ But this subset of MD syntax applies everywhere!

Exercise: write your README

- Compose the README of your software/codebase by making a Markdown file called `README.md` and located in the top level of your codebase folder
- Here are two more example README files: Pluto.jl, Manopt.jl
- ***Allocated time: ~10 minutes!***
  - remember: that is not enough for a polished summary
  - we are building a foundation!

Documentation Strings

Documentation strings

- ***Documentation strings (docstrings)*** are descriptions of what a function does and how it can be used. They ***form the basis of a software documentation!***
  - In Julia they are the topmost string above a function's source code, in Python they are the first string in the function. In both cases they utilize Markdown syntax!
- They are written in human language, but they are ***not*** comments
- They are ***superior*** to comments and should be preferred when possible
- They are accessible from within the language, interactively. Example: `?findall`
- ***Public API should be defined entirely on the docstrings of the exported names***

T&T: First write the docstring, then write the function (describe what it should do before coding it!)

Good documentation strings examples

- Showcase 1: approximate entropy, type
  - rendered docstring
  - source docstring

- Showcase 2: moving agents, function + multiple dispatch
  - rendered docstring
  - source docstring 1, 2

How to write good docstrings

- Clear call signature in code syntax
  - Should only include only the most important information, not every keyword!
  - Multiple call signatures or docstrings may be used for multiple dispatch functions.
- Brief summary of the functionality
- What are the input(s) and output(s) and their Types if not obvious
- Cross-references to related functions if sensible
- Keyword arguments list if the function has some
- Detailed description of functionality if function is scientifically involved
- Citations to relevant scientific papers if any
- [Optional] Example code illustrating usage (I prefer runnable examples!)

T&T: Be clear what you are referring to

- Or, “Use unique identifiers”, or “Pick One Word Per Concept”
  - *Clean Code – A Handbook of Agile Software Craftmanship*
- When referring to the same things, use the same words. ***Exactly the same.***
  - This is a good advice for any form of technical writing (paper, docs, code)
- This establishes ***an unambiguous connection*** between the English words, and the object they refer to, which is typically code
- It is much more important for a technical document to be clear and precise, than to be poetic. ***Do not optimize word variety, optimize word clarity.***
  - E.g.: Fetch, Retrieve, Get, Obtain… Pick one!
  - E.g.: Sketch, Diagram, Concept, Idea... Pick one!
- When referring to an object ***in code***, such as a function or an input to a function, use backticks (`) to convey that the object exists in the code! (through markdown) `this is code text rendered in a monotype font`

Markdown enhanced for Documentation

- `` [`some_function`](@ref) `` links to some function/type with exactly this name

- `[some section](@ref section_id)` links to some section in the documentation that has been tagged as "section_id" (will learn later how to tag)

- `[SomeCitation1984](@cite)` links to a BiBTeX citation entry with same tag
  - ignore this if you don't have any citations (why though?!)

- Why? These special syntaxes ***will become hyperlinks*** when we create/compile the documentation into HTML (last parts of this tutorial)
  - Example again: (source code) (rendered docstring)

Exercises: Documentation strings

- Write ***good*** documentation strings for 5-10 functions of your software
  - Utilize the last 4 slides for how to do so!

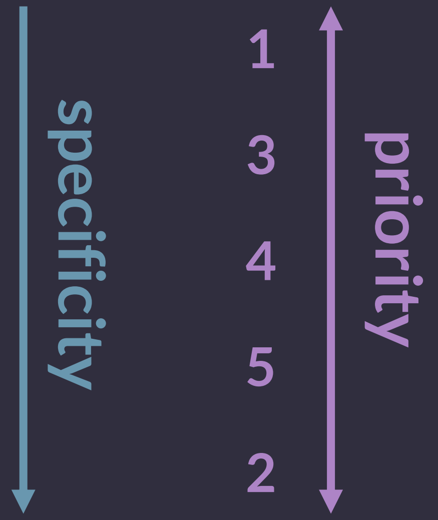
- ***Allocated time: ~20 minutes!***

Overarching Tutorial

Got the basics down… now what?

Recall the structure of a good documentation (***several levels of depth of exposition***)

- ❑ **Highest level**: Summary of what the software does ✓
- ❑ **High level**: One overarching tutorial on the core of the software
- ❑ **Intermediate level**: Examples
- ❑ **Low level**: internals, explanations, how-tos
- ❑ **Lowest level**: Declaration of pubic API / documentation strings ✓



Now it’s time for the overarching tutorial!

And for this, we need Documenter.jl!
(Python users would use Sphinx)

Overarching/Introductory Tutorial/Guide

- Tutorial ≠ explanation! Tutorial = demonstration!
  - It's not about how the software works, it's about how to use it!

- ***Tutorial = concise but typical workflow utilizing all major aspects of the software***

1. Opening statements: what is this tutorial about, what are we expected to learn, what is the typical workflow
2. main inputs and how to create them; introduce new types provided or specific data expected
3. typical/key functions provided by the software that utilize the inputs; showcase a small variety
4. Main outputs and how to use them; typical analysis and/or visualizations
5. More variability of inputs → key functions → outputs → typical analysis
6. links to the rest of the docs: the API, where to learn more, explanations, links to other package's docs, …

- ***Absolutely crucial: tutorials and examples showcase runnable code and are RUN!***

Good overarching tutorial examples

- Attractors.jl, Agents.jl, ModelingToolkit.jl, DynamicalSystems.jl, Documenter.jl, ComplexityMeasures.jl, Turing.jl, Makie, GeoStats.jl, QuantumOptics.jl, DrWatson

T&T: at the start of the tutorial have a copy-pasteable version of the tutorial in a single code block (with guiding comments), if possible

Documenter.jl + Literate.jl

Documenter.jl: popular package for creating documentation for Julia packages & deploying online

- Enhances markdown syntax with many capabilities such as cross-referencing sections and docstrings
- ***Executes Julia code blocks and interlaces them with their output***
- Checks that the documentation is working properly
- Lots of automation: t.o.c., expanding docstrings and more
- Compiles a documentation (HTML, PDF, …) using ***markdown files***
- ***Best approach for tutorials/examples: Script+Comments -> Literate.jl -> Documenter.jl***

Literate.jl: converts runnable Julia scripts into Documenter markdown files (and other formats)

More Documenter.jl Markdown enhancements

- When writing docstrings we encountered already some Markdown enhancements:
- ``[`some_function`](@ref), [some section](@ref tag), [SomeCitation1984](@cite)``

Markdown file for Documenter

~~~
# [Section](@id tag)
                   └→ tagged section

```@index          ──→ lists and hyperlinks to all
```                    expanded docstrings

## [Docstrings](@id docs)

```@docs           ──→ expands docstrings
MyCodebase
bar(::Int)
foo                ──→ [`foo`](@ref)
```                    will link here
~~~

~~~
```@example main   ──→ example block: will show
using MyCodebase       code, and then its output
x = rand(10)           in the compiled docs
y = foo(x)         ──→ output = last statement
~~~

```@example main   ──→ name of block == same →
using CairoMakie       same environment

scatter(x, y)      ──→ return figure = show
```                    figure in compiled docs
```

when we compile/build the docs with Documenter.jl,
the special syntax in this Markdown file will be replaced with what it is supposed to be!
```

# Script → Literate.jl → Documenter.jl Markdown

Instead of creating a markdown file:

~~~
# [Tutorial](@id tutorial)

For example, we use [`foo`](@ref)
```@example main
using MyCodebase
y = foo(1, 2, 3)
```

from which we get
```@example main
choose plotting backend
using CairoMakie
density(y)
```

Have a look at the functions:
```@docs
foo
```
~~~

markdown ⟷ comments

@example ⟷ code snippet

create a Julia script with comments:

````
# # [Tutorial](@id tutorial)

# For example, we use [`foo`](@ref)
using MyCodebase
y = foo(1, 2, 3)

# from which we get
## choose plotting backend:
using CairoMakie
density(y)

# Have a look at the functions:
# ```@docs
# foo
# ```
````

start line with ## for comment in code

Convert from script to markdown (example: file, conversion)

```
Literate.markdown(".../tutorial.jl", ".../outpath")
```

T&T: Word is gold

weigh every sentence in the docs in gold

- ❑ Don't make the tutorial longer than necessary!
- ❑ Include the most important things, and, only if there is space, go into secondary details.
- ❑ Be concise and precise, avoid repetitions. Re-reading your docs helps!
- ❑ The documentation is about the software. Don't include information like "how to use text editors". This is something for lectures or course material.

conciseness

every sentence should be worth its weight in gold

- ❑ The words in the docs should be gold. Doc pages should be very well written. Given that you have finite time, decide what are the most important things to include in the documentation, and prioritize writing those in high quality.
- ❑ Low quality can be worse than nonexistence: confusing, misleading, decrease trust…
- ❑ Pages that are low quality could be better off not being included at all until ready!

quality

Exercise: write your own Tutorial!

- Create a Julia script that will be your software's Tutorial
  - Following Literate.jl -> Documenter.jl approach and utilize Documenter.jl Markdown in comments

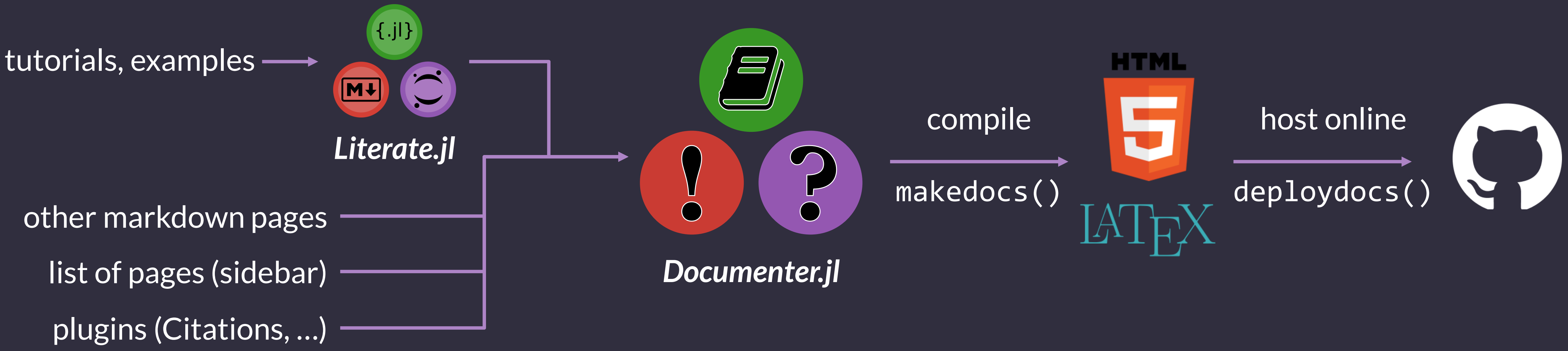
- Introduce the key functions. Create their inputs and show their outputs by making `` `@example` `` blocks (which is normal Julia code in the script that will be passed to Literate.jl). Then, also expand the docstrings of these key functions using `` `@docs` `` block (which will be comments in the Julia script given to Literate.jl)
  - In the next section where we will compile to HTML: examples will be run, links will be resolved

- Follow the advice of the last 5 slides and write a ***draft*** for your tutorial!
  - We won't have the time to write the whole thing! Good docs take many hours!
  - But we can write an outline and give feedback on that!!!

- ***Allocated time ~ 30 minutes!***

Compiling your documentation

Typical Documenter.jl workflow



Folder layout and the “make.jl” file

Folder structure:

Julia, optional

```
MyCodebase/
├── test/runtests.jl
├── src/
│   ├── ...
│   └── MyCodebase.jl
├── Project.toml
└── docs/
    ├── src/
    │   ├── index.md
    │   ├── api.md
    │   └── tutorial.jl
    ├── make.jl
    └── Project.toml
```

Documenter

documentation dependencies (arbitrary, independent` of formal package dependencies)

make.jl file

```julia
cd(@__DIR__) # go into `docs` folder
import Pkg; Pkg.activate(@__DIR__)
using Documenter, Literate, MyCodebase
# convert tutorial/examples to markdown
Literate.markdown("./src/tutorial.jl", "./src")
# Which markdown files to compile to HTML
# (which is also the sidebar and the table
# of contents for your documentation)
pages =  [
    "Introduction" => "index.md", mandatory
    "Tutorial" => "tutorial.md",
    "API" => "api.md",
]
# compile to HTML:
makedocs(; pages, modules = [MyCodebase], kw...)
```

DocumenterCitations.jl

- Wonderful extension to Documenter that allows you to cite articles via standard BiBTeX syntax. ***Uber cool for scientific software!***
- Enables the `[Citation](@cite)` syntax in markdown (docstrings, or .md pages)
- To use, install DocumenterCitations, and adjust:

1. Add your BiBTeX .bib file in `docs/src/refs.bib`
2. Add this block in any documentation page (under a `` `References` `` section):
   ~~~
   ```@bibliography
   ```
   ~~~
3. Adjust your `make.jl` file and `makedocs` command like so:

```julia
using DocumenterCitations
bib = CitationBibliography(joinpath(@__DIR__, "src", "refs.bib"))
makedocs(; pages, plugins = [bib], kw...)
```

Exercise: compile your 3-pages Documentation!

- Following the example `make.jl` file of the previous slide, build your documentation with only two pages: a mandatory opening page (homepage) named `index.md`, and the Literate-converted tutorial.
- Run the `make.jl` file. `makedocs` will error when things don't work: incorrect references, errors in the example code snippets, …
- Continue solving these errors until `makedocs` finishes building the HTML pages! If you can't, pass the keyword `warnonly = true` to `makedocs`, so that it complains instead of erroring
- Then, a folder `docs/build` will exist locally. Go there and open the `index.html` file!
- ***Allocated time ~ 30 minutes!***

for now use this short example as your "index.md" file, which uses simply copies the README file (see first T&T)

~~~
# MyCodebase

```@docs
MyCodebase
```

See the [Tutorial](@ref tutorial)
for how to use this package!
~~~

and use this trival example as the "api.md" file

~~~
# API
```@docs
function1
function2
other_docstrings_you_wrote
```
~~~

Deploy/host docs on GitHub

Deploying docs

- Deploying your documentation to a GitHub repo is “super easy”!

- You augment your `make.jl` file and at its end add

```
deploydocs(
    repo = "github.com/USER_NAME/REPO_NAME.git",
)
```

- That’s all that’s necessary from the Julia/Documenter.jl side!
- Life is so damn simple!

Hosting on GitHub

- GitHub provides (for free) hosting, and allows deployment via GitHub actions

- Live demonstration of building and deploying docs for a package of JuliaDynamics

- Example file of the GitHub action ; copy this!
- It uses the docs/Project.toml file to get the dependencies!
  (the line `--project=docs/` in the CI file)

- When you are done push your codebase to your GitHub repo
  - In truth, we only care about the `docs` folder and whatever dependency the docs have

Exercise: host your docs on GitHub

Things may get a bit finicky; keep calm and ask for assistance if things don't work

- You need a GitHub account, and a GitHub repo that hosts your `docs` dir.
- You enable GitHub actions by copying the provided .TOML file into the directory https://...YourRepo/.github/workflows/documenter.yaml
- .TOML file is here (in block5 in GoodScientificCodeWorkshop)
- Once you push any commit to the GitHub repo (to trigger the GitHub actions for the first time) the documentation build and deployment will start...
- Follow closely the log of changes by clicking on the "Actions" tab of your GitHub repo, and see if any errors occur
- If the build fails, we will help you!!! 😰 😱 😨 🙀
- ***Allocated time ~ 15 minutes!***

Congratulations! You are the best! You graduated the docs academy! Only things that remain now are more tips & tricks, and for you to enrich your documentation with more pages, more examples, and polish further!

T&T: The one true ~~Morty~~ Documentation

- You want to eliminate chances of “forgetting to update something”. Two rules:

1. Documentation ***must exist only at one place***
   - GitHub webpage hosting, READMEs, GitHub Wiki, Private Webpages, GitLab pages, …
   - ***NO!*** Keep it in ***ONE*** place, coherently, and allowing cross-referencing everything!
   - That’s also why I do ***not*** like example code in docstrings ***and*** in runnable examples! ***Choose one!***

2. Thing X must be ***defined only once, at one place***, and documented at that place in your documentation with as much detail as necessary!
   - All other parts of the documentation that use X must cross-reference its definition
   - Not explain it again! Avoid duplicating information!
   - Cross reference properly via hyper-link creation using your documentation tool (Documenter.jl)!

PRAISE THE ONE TRUE ~~MORTY~~ DOCUMENTATION

What to put in the Homepage (index.md)

- Start homepage with the README (= module docstring, see first T&T)
- Then, add the following information if it isn't already in the README:

1. Logo!
   - Makes your software unique, memorable.
   - Shows you care for it enough to give it a "face".
2. Getting started
   - how to install, preliminary knowledge, how to navigate the documentation
3. List of features (if too extensive, can be a dedicated page)
4. Design philosophy (optional)
   - More in-depth description of the software design, what makes it special
   - Can also include a comparison with competing software
5. How to contact the authors / ask questions / get involved in development
6. Limitations / caveats

ToC: Table Of Contents (sidebar)

- ***The Eye Horizon***
  - The viewer will scan the page content currently in front;
  - This scan should indicate if the information the user needs is in this page
  - ***If not***, this scan should also indicate ***which other*** page the information is in

- The Table of Contents is the best way to satisfy the “eye horizon” requirement!
  - Requires you to structure the doc page around headers and subheaders

- Get header ordering right!
  - Subheaders must really be subtopics of the upper level
  - Same level of information variety must be same header level
  - Sweet-spot balance between overview and information exposure

T&T: Read your docs

- As for the last tip, to make the best docs you can… ***read your docs!***
- ***Read them. Again and again!***
- And while doing so, and ask yourself critical questions, like:
  - Is this docstring clear and satisfy the key 5-6 components of this tutorial?
  - Are the key interfaces exposed clearly and concisely, with exemplary input and output?
  - If I knew nothing about this software, would the tutorial be enough for me to use it?
  - Are there any typos lingering around?
  - What is the meaning of life?
  - Are there algorithms I describe without referencing their associated journal article?
  - What will I have for dinner tomorrow?
  - Oh man I need more coffee… :D

Good Documentation examples

We close the workshop by showcasing docs of various packages that community members consider “exceptional”. Use them to get inspired!

Agents.jl, Oceanigans.jl, DifferentialEquations.jl, DrWatson, JuMP, DynamicalSystems.jl, Manopt.jl, DocumenterCitations.jl, FastAI.jl, DataFrames.jl, GeoStats.jl, QuantumOptics.jl, …

and of course, the Julia manual itself, which is so good you typically don’t need a textbook to learn Julia

Exercise: Continue working on the docs!!!

- Use the remaining time to improve the docs!
- Write more docstrings
- Improve/complete the tutorial
- Add more examples
- Add explanations / internals pages
- Present your docs to your peers
- Enjoy your new life as an exceptional documenter ;)