Teaching workflow systems to new **graduate students**: why is it so hard?

Workflow systems as mirrors of computing complexity.

C. Titus Brown
School of Veterinary Medicine;
UC Davis
ctb on github; ctbrown@ucdavis.edu; titus.idyll.org on bluesky.

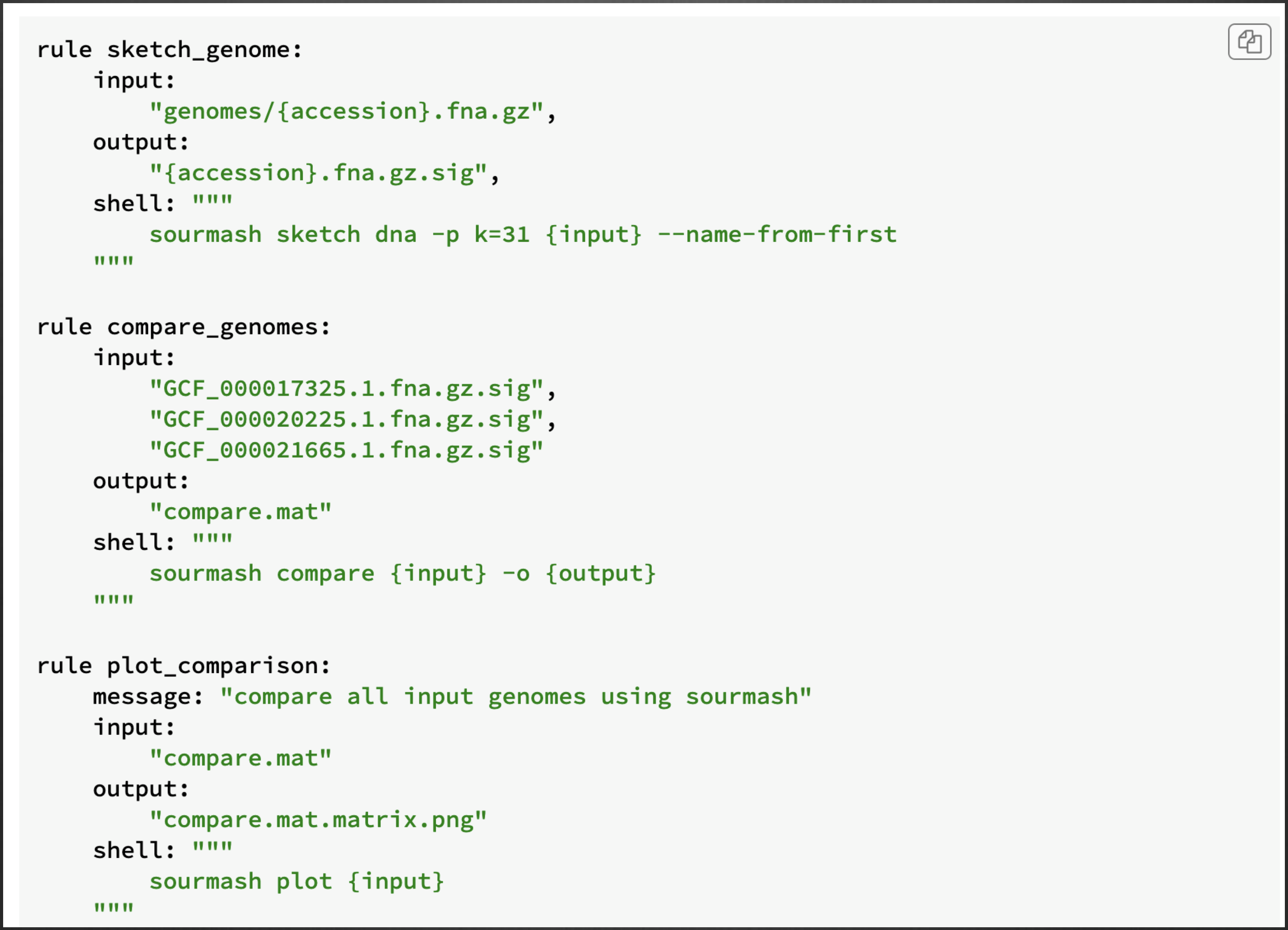
UCDAVIS VETERINARY MEDICINE

Sep 24, 2025

Talk slides available! google 'titus talk osf.io'

Snakemake in two slides

1. Rules & wildcards ({accession})



```
rule sketch_genome:
    input:
        "genomes/{accession}.fna.gz",
    output:
        "{accession}.fna.gz.sig",
    shell: """
        sourmash sketch dna -p k=31 {input} --name-from-first
    """

rule compare_genomes:
    input:
        "GCF_000017325.1.fna.gz.sig",
        "GCF_000020225.1.fna.gz.sig",
        "GCF_000021665.1.fna.gz.sig"
    output:
        "compare.mat"
    shell: """
        sourmash compare {input} -o {output}
    """

rule plot_comparison:
    message: "compare all input genomes using sourmash"
    input:
        "compare.mat"
    output:
        "compare.mat.matrix.png"
    shell: """
        sourmash plot {input}
    """
```

2. ‘expand’ to generate targets (“concrete” rules)

```
ACCESSIONS = ["GCF_000017325.1",
              "GCF_000020225.1",
              "GCF_000021665.1",
              "GCF_008423265.1"]

rule sketch_genome:
    input:
        "genomes/{accession}.fna.gz",
    output:
        "{accession}.fna.gz.sig",
    shell: """
        sourmash sketch dna -p k=31 {input} --name-from-first
    """

rule compare_genomes:
    input:
        expand("{acc}.fna.gz.sig", acc=ACCESSIONS),
    output:
        "compare.mat"
    shell: """
        sourmash compare {input} -o {output}
    """
```

Snakemake is the best workflow system for biology.

It’s got a big community and a lot of users. Just use snakemake.

(Don’t @ me.)

Ok but really:
Snakemake is not a bad workflow system choice for biologists.

- It’s got a big community and a lot of users.
- It’s integrated with Python and so you can extend your Snakemake workflow with the best* programming language.
- It’s fantastic for exploratory research workflows!!
- I would also advise people look at nextflow.
- CWL and WDL are excellent production options.

* well, also see Rust.

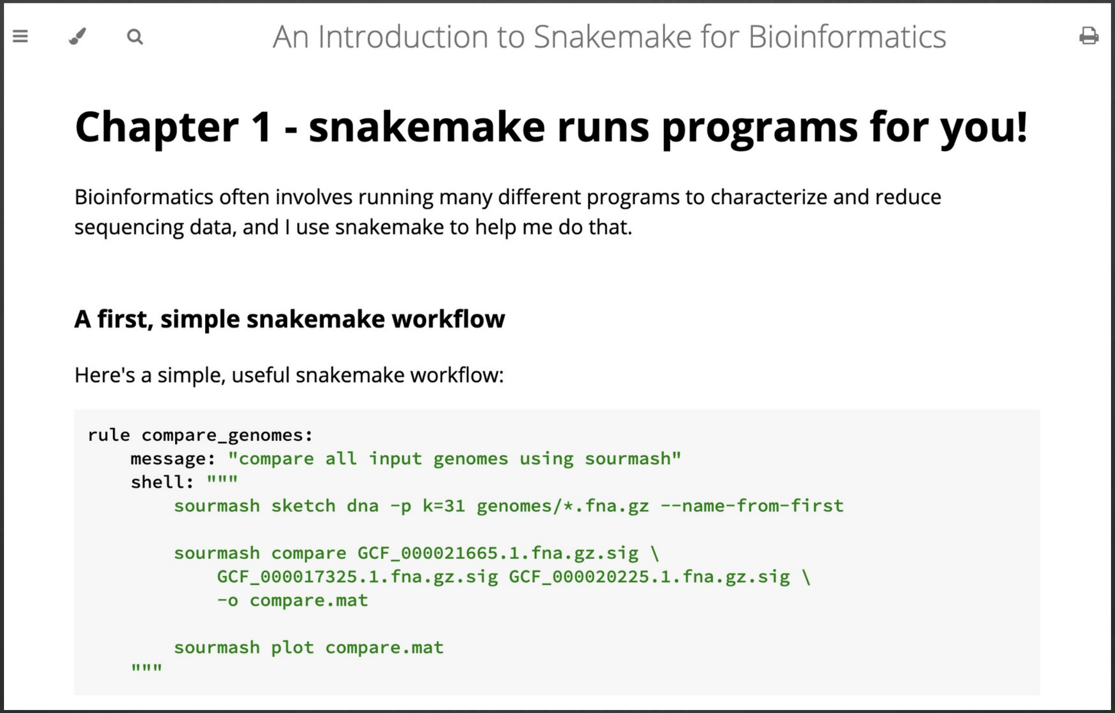
Bernie
I am once again asking
for you to embrace snakemake
imgflip.com

One big problem with snakemake:

It’s hard to learn, and the documentation is good, but only if you’re already computationally expert.

One big problem with snakemake: it's hard to learn

Solution?



Chapter 1 - snakemake runs programs for you!

Bioinformatics often involves running many different programs to characterize and reduce sequencing data, and I use snakemake to help me do that.

A first, simple snakemake workflow

Here's a simple, useful snakemake workflow:

```
rule compare_genomes:
    message: "compare all input genomes using sourmash"
    shell: """
        sourmash sketch dna -p k=31 genomes/*.fna.gz --name-from-first

        sourmash compare GCF_000021665.1.fna.gz.sig \
            GCF_000017325.1.fna.gz.sig GCF_000020225.1.fna.gz.sig \
            -o compare.mat

        sourmash plot compare.mat
    """
```

https://ngs-docs.github.io/2023-snakemake-book-draft/

Introducing me and my motivations!

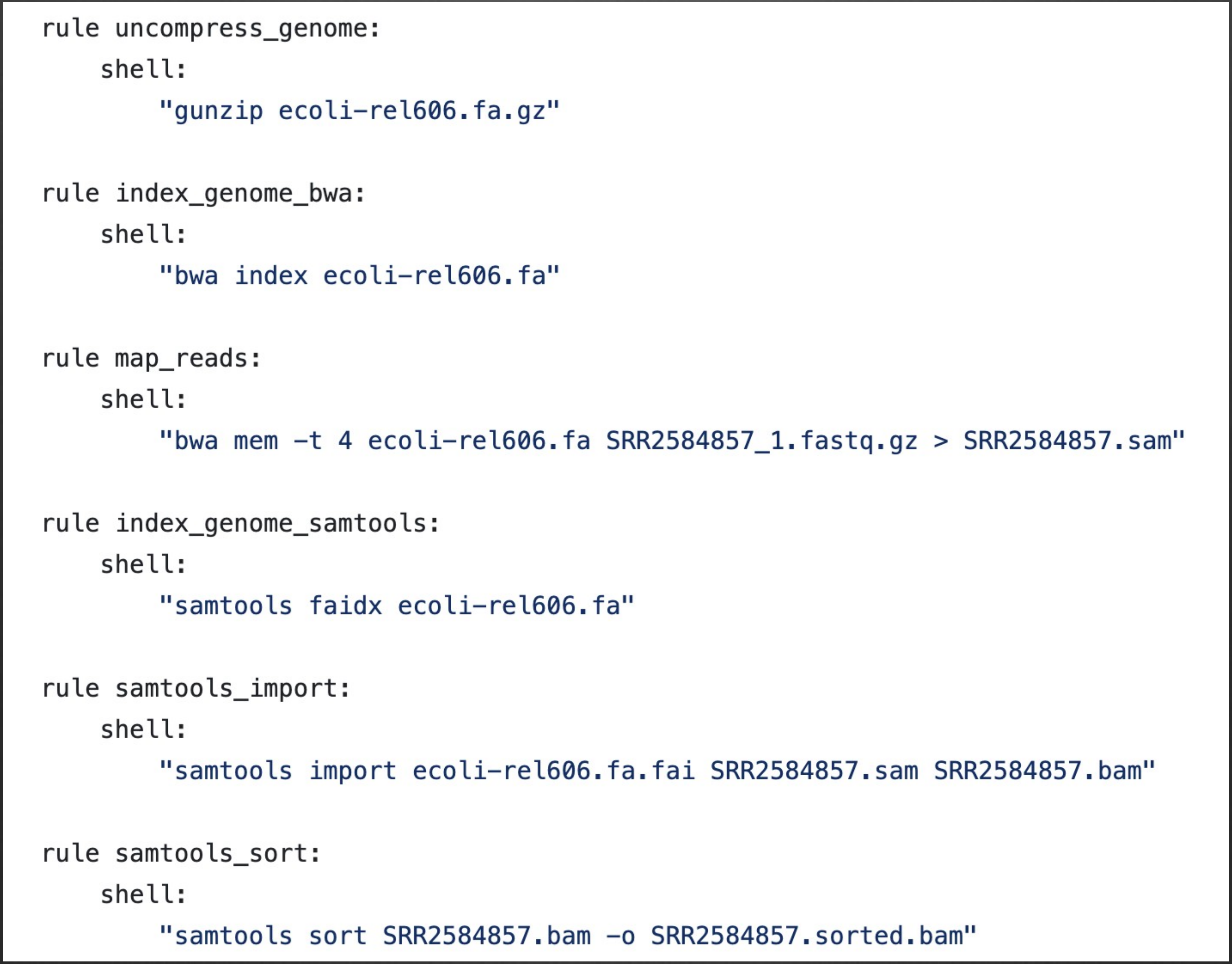
- Professor at UC Davis. One of the “bioinformatics generalists” on campus. Many of the biologists need to do bioinformatics at scale; I would like to help them.
- Separately, I am an open sourcenik, with decades of experience openly developing and maintaining software. Really interested in good open online community.
- Currently the support and maintenance dude for sourmash, trying to answer the question: “Can a senior prof also be an open source maintainer?”
- Anyway, back to teaching snakemake…

Two approaches I've used to teaching snakemake

1. Inductive. Start with some shell commands, and slowly decorate and generalize as we go.

2. Deductive. Build a workflow progressively.

Approach 1: Start with a bunch of shell commands:



```
rule uncompress_genome:
    shell:
        "gunzip ecoli-rel606.fa.gz"

rule index_genome_bwa:
    shell:
        "bwa index ecoli-rel606.fa"

rule map_reads:
    shell:
        "bwa mem -t 4 ecoli-rel606.fa SRR2584857_1.fastq.gz > SRR2584857.sam"

rule index_genome_samtools:
    shell:
        "samtools faidx ecoli-rel606.fa"

rule samtools_import:
    shell:
        "samtools import ecoli-rel606.fa.fai SRR2584857.sam SRR2584857.bam"

rule samtools_sort:
    shell:
        "samtools sort SRR2584857.bam -o SRR2584857.sorted.bam"
```

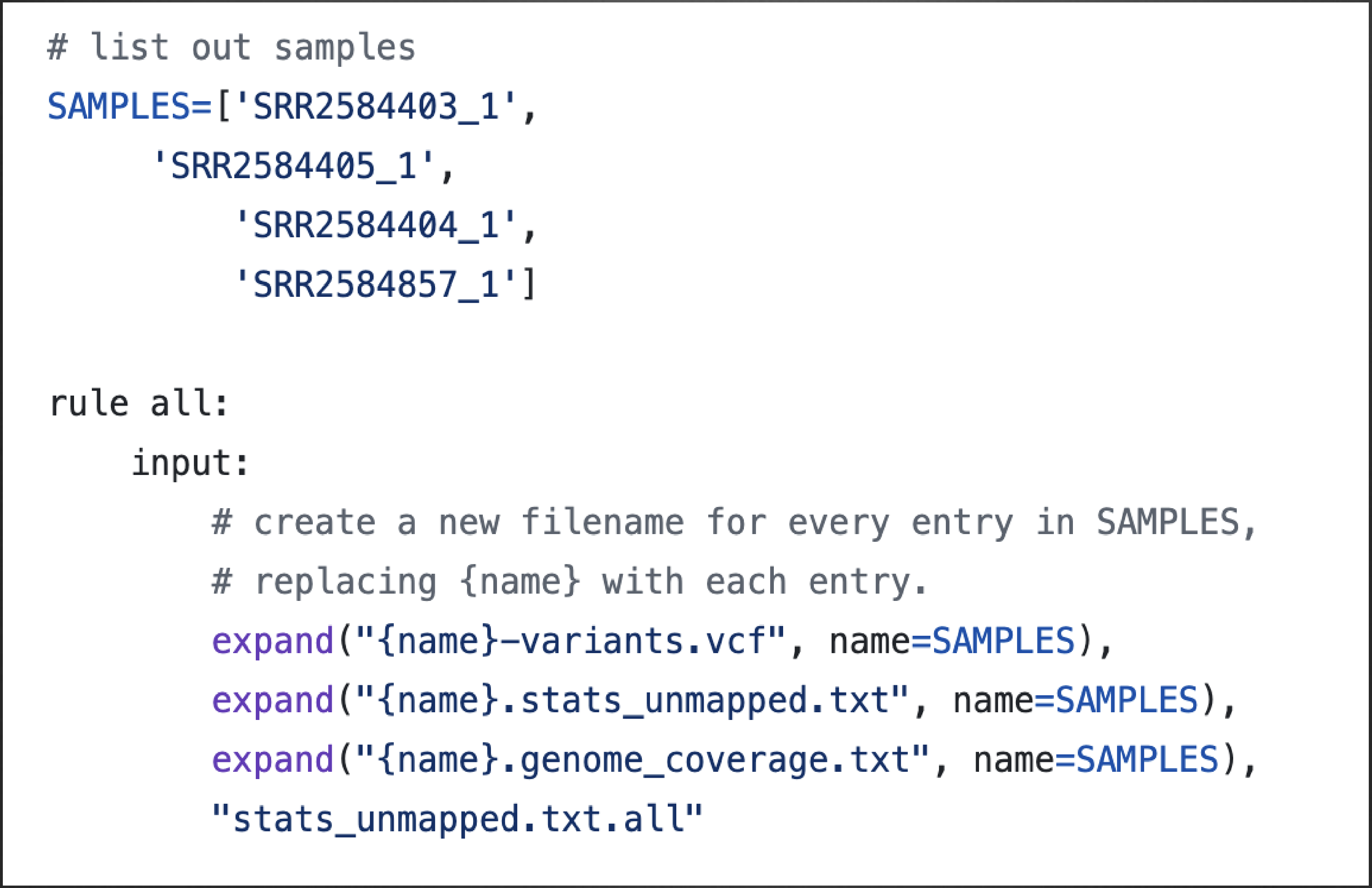
Decorate as we go:

```
rule uncompress_genome:
    input: "ecoli-rel606.fa.gz"
    output: "ecoli-rel606.fa"
    shell:
        "gunzip ecoli-rel606.fa.gz"

rule index_genome_bwa:
    input: "ecoli-rel606.fa"
    output:
        "ecoli-rel606.fa.amb",
        "ecoli-rel606.fa.ann",
        "ecoli-rel606.fa.bwt",
        "ecoli-rel606.fa.pac",
        "ecoli-rel606.fa.sa"
    shell:
        "bwa index ecoli-rel606.fa"

rule map_reads:
    input:
        "ecoli-rel606.fa.amb",
        "SRR2584857_1.fastq.gz"
    output:
        "SRR2584857.sam"
    shell:
        "bwa mem -t 4 ecoli-rel606.fa SRR2584857_1.fastq.gz > SRR2584857.sam"
```

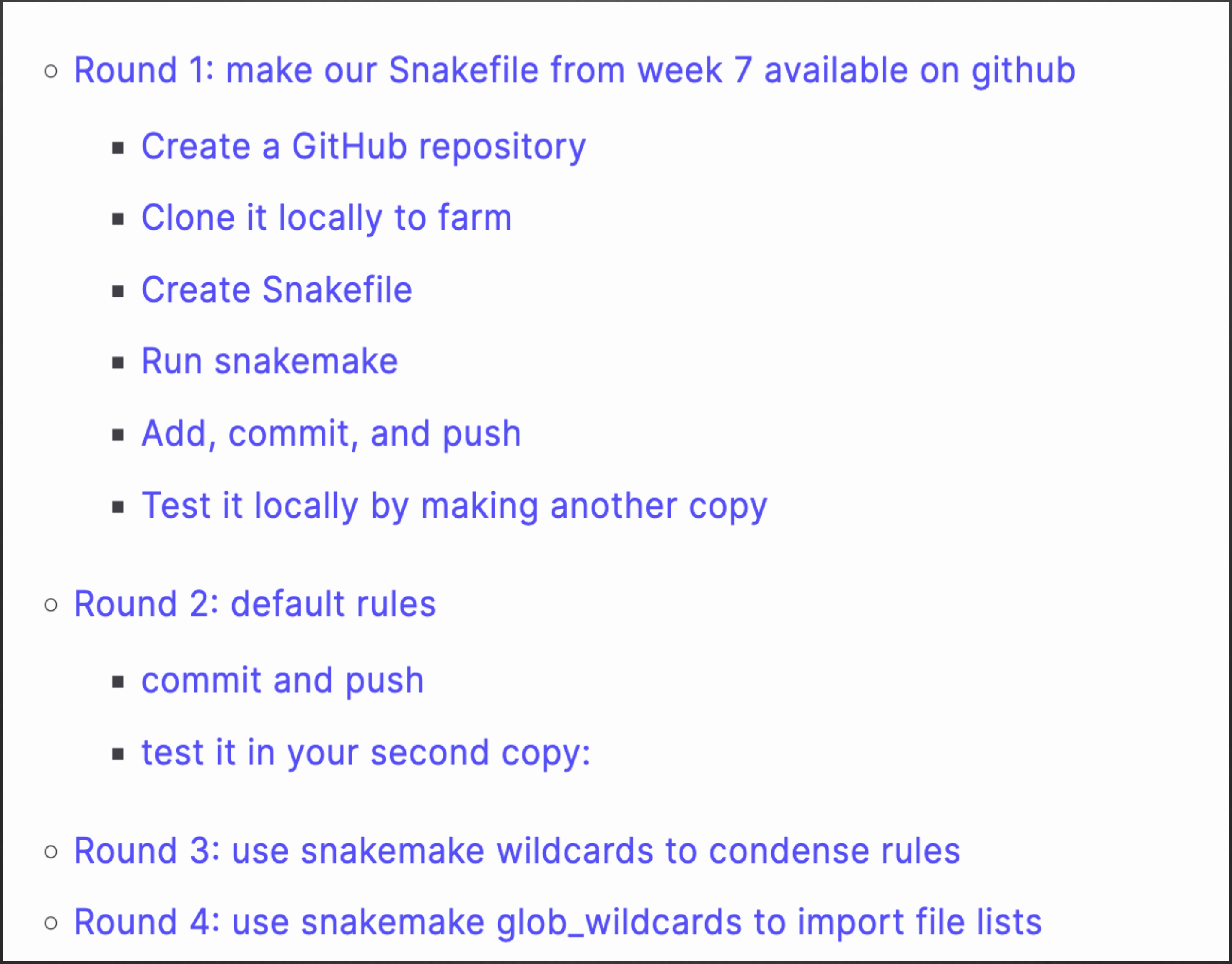
End by demonstrating the power of this fully functional workflow:



```
# list out samples
SAMPLES=['SRR2584403_1',
     'SRR2584405_1',
         'SRR2584404_1',
         'SRR2584857_1']

rule all:
    input:
        # create a new filename for every entry in SAMPLES,
        # replacing {name} with each entry.
        expand("{name}-variants.vcf", name=SAMPLES),
        expand("{name}.stats_unmapped.txt", name=SAMPLES),
        expand("{name}.genome_coverage.txt", name=SAMPLES),
        "stats_unmapped.txt.all"
```

Approach 2: Start with basic rules, explain as we go



- Round 1: make our Snakefile from week 7 available on github
    - Create a GitHub repository
    - Clone it locally to farm
    - Create Snakefile
    - Run snakemake
    - Add, commit, and push
    - Test it locally by making another copy
- Round 2: default rules
    - commit and push
    - test it in your second copy:
- Round 3: use snakemake wildcards to condense rules
- Round 4: use snakemake glob_wildcards to import file lists

Do either of these approaches work?

- Not clear? Some people seem to get it… 🤷
- Teaching grad students this stuff is really hard.
- Moreover, it takes *practice* to get good at this, and bio grad students usually don’t actually need to do real bioinformatics until their 3rd year, when they no longer want to take classes.
- This is the best I can do in the first year, and every now and then a 3rd year student comes back and says “ohmigod I get it now! You weren’t actually a terrible teacher!”

Why is this so hard??

Theory 1: workflow systems are a mirror of our computing landscape

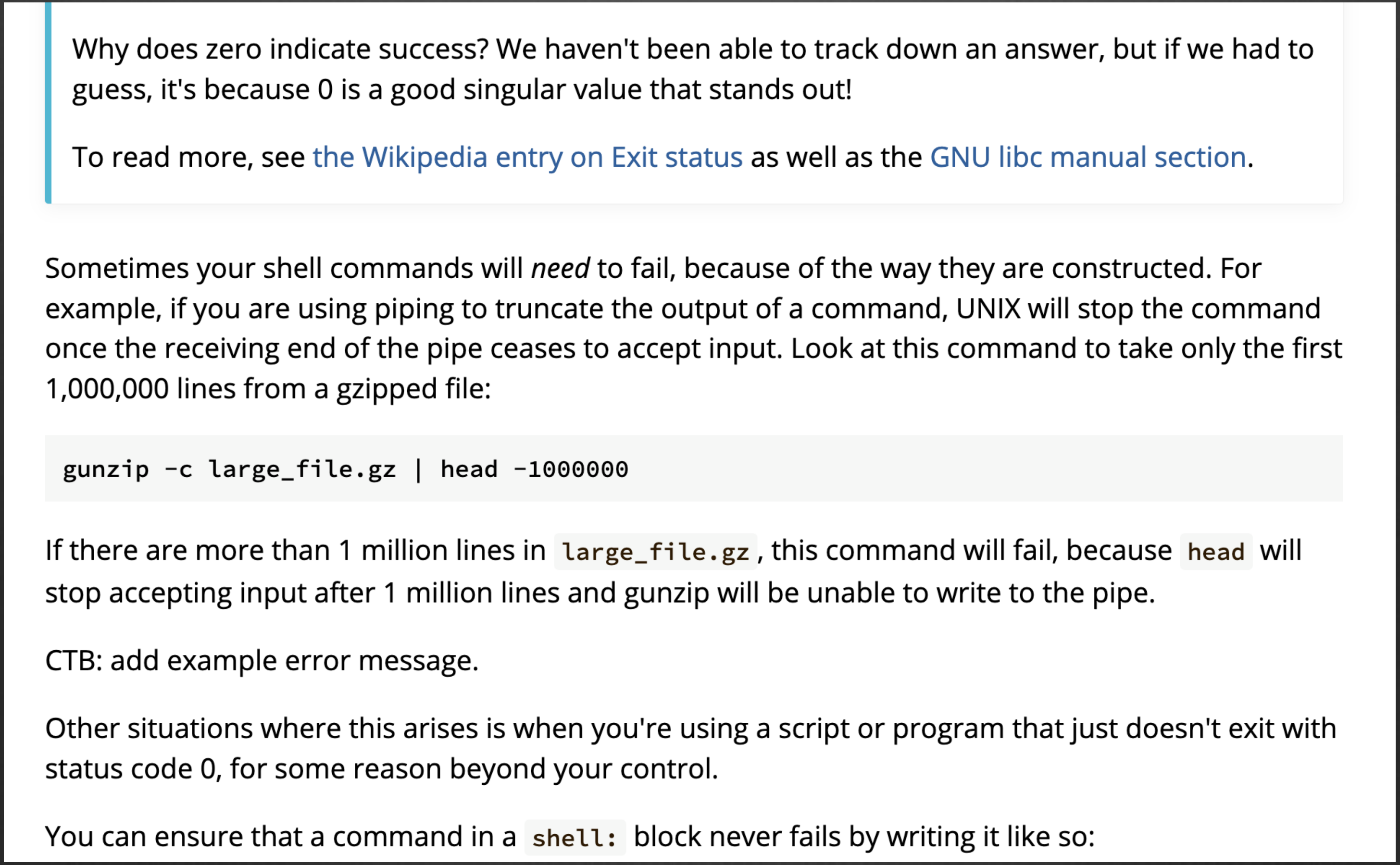
- …which is complex and capricious.
- Almost every feature in our workflow systems exists because of a choice made in how our computing infrastructure actually works (or, well, fails ;)

My favorite example: exit codes!

- A really important feature of workflow systems is telling you when a job fails.
- By default, our UNIX shells don't visibly complain when a job bombs, and a lot of software doesn't clearly signal when a catastrophe happens.
- (How many of you have written a shell script that just blithely keeps on going when an important command fails?)
  - YES, I know about 'set -e', but it's not on by default! For good reasons!
- This is a *huge* problem for newbies!

  "I got 1,000 lines of output... did my script succeed or fail?!"
- Snakemake deals with this by *forcing* failure if exit code is not zero. This turns out to be one of the biggest deals about it, for me 😆

(Why zero? Who the heck knows!)



Why does zero indicate success? We haven't been able to track down an answer, but if we had to guess, it's because 0 is a good singular value that stands out!

To read more, see the Wikipedia entry on Exit status as well as the GNU libc manual section.

Sometimes your shell commands will *need* to fail, because of the way they are constructed. For example, if you are using piping to truncate the output of a command, UNIX will stop the command once the receiving end of the pipe ceases to accept input. Look at this command to take only the first 1,000,000 lines from a gzipped file:

```
gunzip -c large_file.gz | head -1000000
```

If there are more than 1 million lines in `large_file.gz`, this command will fail, because `head` will stop accepting input after 1 million lines and gunzip will be unable to write to the pipe.

CTB: add example error message.

Other situations where this arises is when you're using a script or program that just doesn't exit with status code 0, for some reason beyond your control.

You can ensure that a command in a `shell:` block never fails by writing it like so:

https://ngs-docs.github.io/2023-snakemake-book-draft/recipes/never-fail-me.html

UNIX is somewhat arbitrary and capricious…

- …because choices needed to be made, and they were made, and now we're dealing with the consequences of those choices ~60 years later.
- (And maybe in another 1000 years, too – ref Vernor Vinge)
- This is a challenging thing to teach.

(YES, there is a reason for many of these decisions. YES, they are not necessarily bad decisions. I'm just saying that you can't figure them out from first principles, and so they need to be taught in some way! And YES, there are deep principles underlying computing. Now go try teaching it to a newbie.)

Implications of theory 1:

“Workflow systems are a mirror of our computing landscape”

Maybe, just maybe, if we could teach a workflow system properly and thoroughly, people would come away a more in-depth understanding of how computers work.

Maybe, a really good book on snakemake could be a really good book on practical computing.

Another challenge: how do we decide what order to teach things in?

There is basically no principled approach to teaching practical computation, as far as I can tell.

No concept inventory, no standard textbook, etc.

(YES, I’m sure you have your opinions. 😆)

So what order do you introduce concepts in??

Maybe teach based on what people already know?

But this turns out to be heavily path dependent and hence individualized.

Theory 2: we should design computational training based on (what I have named) Kernighan's Principle

Kernighan's *Law:*

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

-- Brian Kernighan, 1974

(I was born in 1974. WTF.)

This ”law” led me to “Kernighan’s Principle for Lesson Design”:

Teach practical computational concepts in the order of their *debuggability*.

Running shell commands? Easy, copy paste the command – does it work?

Wildcards? Easy, snakemake will output the rewritten rule for you with –p.

Expand? Easy, you can debug with ‘print’.

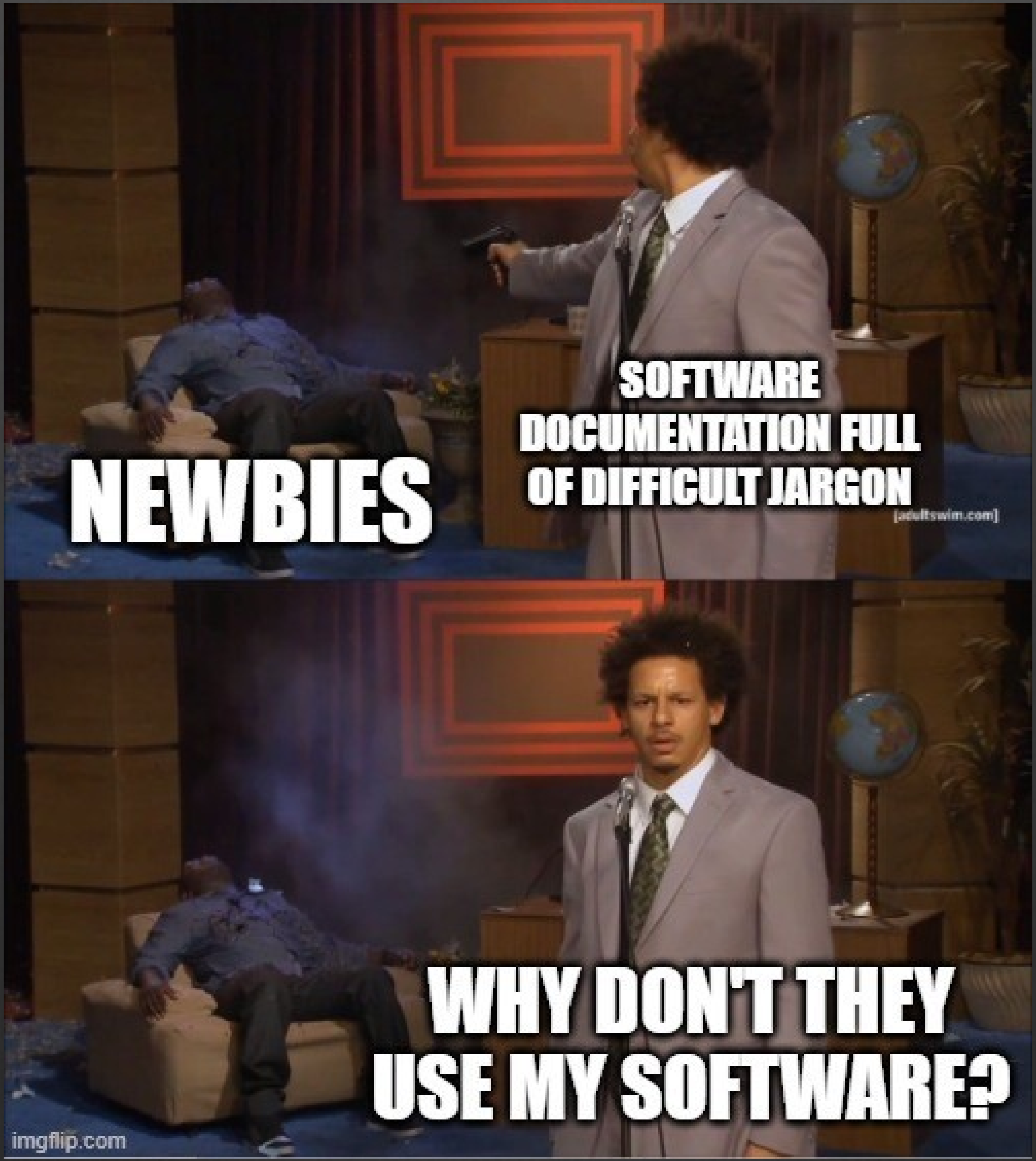
Cluster distribution? Feckin’ impossible to debug.

Concluding thoughts for discussion ☺

1. Teaching workflow systems is hard because you essentially have to teach an abridged version of ALL OF COMPUTING.

2. Teach practical computational concepts in the order of their *debuggability*.

I think effective use and creation of workflows is a critical skill for today's researchers to know about, and we need to figure out how to teach it well!


SOFTWARE DOCUMENTATION FULL OF DIFFICULT JARGON
NEWBIES
[adultswim.com]
WHY DON'T THEY USE MY SOFTWARE?
imgflip.com

Thanks to Cassie Olivas for the memes ☺