Science Hacking 101:

Tips and tricks for coding like a *badass* computational scientist

Jeremy R. Manning (jeremy@dartmouth.edu)
Contextual Dynamics Laboratory
Dartmouth College

Simplicity

Simplicity

- **Simplicity**: the art of maximizing the amount of work *not* done.

- When in doubt, ask yourself: “what is the *simplest* way I can approach this part of the project?”

- You may need to design things the messy way first, to understand all of the nuances. Then go back and think about the cleanest/simplest implementation. (Future you will appreciate it!)

- How does the principle of **simplicity** translate into good design practices?

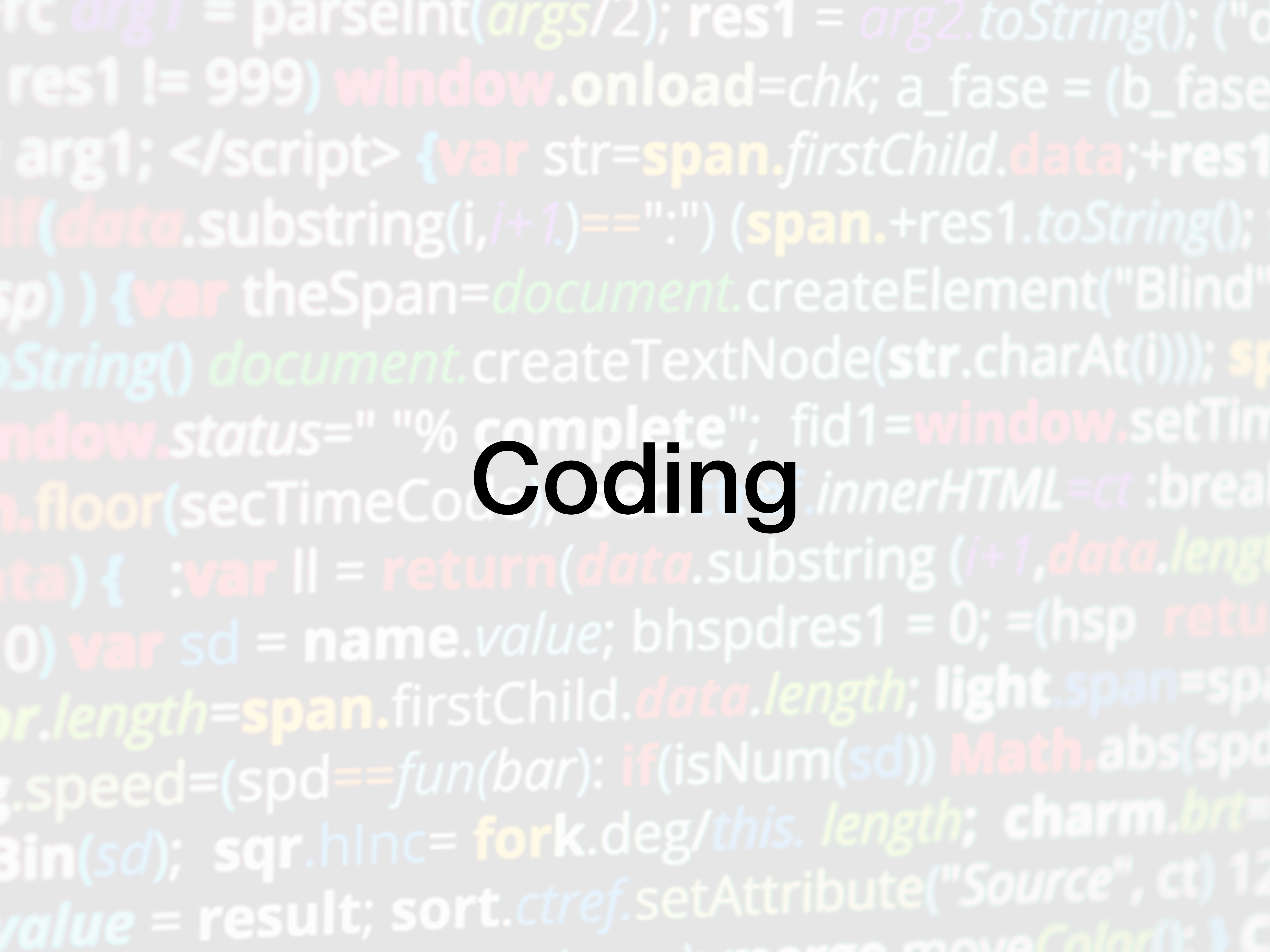
Simplicity: code

- **Write everything once** and re-use
- Consolidate the format of inputs and/or use cases early (***funneling***; more on this later)
- **Fail fast**, before you've wasted the user's time. This also helps with debugging.
- **Minimize work for the *user*** when possible (e.g. set sensible defaults)
- Keep **syntax** and design clean, consistent, and free from clutter
- **Simple** is often better than **fast.** (More on optimization later.)

Simplicity: packaging

- Keep focus **well-defined** and limited in scope. Ask: “what specific problems does my problem solve, and what *doesn’t* it solve?”
  - Note: project scope can (and should!) evolve over time
- Keep folders and organization as flat as possible (more later)
- Modular designs facilitate development, maintenance, testing, and re-use/re-purposing (more later)

Coding



Syntax

- Adhere to the **PEP8** style guide for Python code (or equivalent for other languages, when possible): https://www.python.org/dev/peps/pep-0008/

- **Consistency**: within each type of named object (variables, functions, constants, loop iterators, etc.) use the same naming scheme and style. Keep names simple but descriptive.

- Use **spaces** around operators (e.g. `a += b * c`)

- Keep code **visually clean** by writing short lines and grouping related lines. Goal is to maximize code readability at a glance.

- **Use comments sparingly** but consistently to describe the API (for user-visible functions or complex internal functions) and to describe algorithms (if not obvious)

Syntax: naming

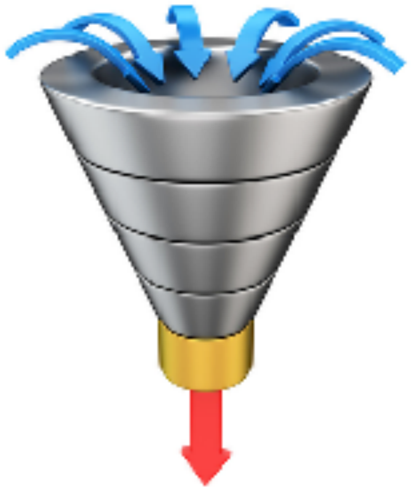
- `x` (single lowercase letter): loop iterators, minor scalar variables
- `X` (single uppercase letter): constants, matrices
- `lowercase`, `lowercase_with_underscores`: variables
- `UPPERCASE`, `UPPERCASE_WITH_UNDERSCORES`: usually constants
- `CamelCase`: classes
- `mixedCase`, `Capitalized_Words_With_Underscores`: no

Syntax: messy code

```
def AddSomeNumbers( my_data ):

  my_sum = 0 #keep track of the sum

  for LoopIterator in range( len( my_data ) ):

    #add the next value to the sum

    my_sum += my_sum+my_data[ LoopIterator ]

  return my_sum
```

Syntax: clean(er) code

```
def sum(x):
  y = 0
  for i in range(len(x)):
    y += x(i)
  return y
```



Funneling

- Writing general purpose (modular) functions often requires supporting a variety of input formats
- To simplify package design, “funnel” data and arguments into a consistent format as early as possible
- This lessens the burden on internal functions and modules, with respect to the number of data formats and options they need to support

Funneling example

```
def brain_plotter(data, *args, **kwargs):

  [data, opts] = format_data(data, *args, **kwargs)

  analyzed = analyze_data(data, opts)

  plot(analyzed)
```

When to split {lines, functions, files, folders}

- **Lines** should be grouped if they are conceptually and/or syntactically related (import statements, performing a related series of calculations on similar data, etc.)

- **Make a separate function** if the group of lines is going to be used by other functions, or if it'll be repeated several times

- **Make a separate file** to organize all functions within a file around a low-level goal or task (e.g. display, i/o, data wrangling, etc.). Caveat: each file should be about a page long. Avoid creating many small files or few very large files.

- **Make a separate folder** to organize files around the same higher level goal (e.g. stats, plotting, interface, etc.). Try to keep the total number of folders small and the organization relatively flat.

Storyboarding

Storyboards

- Describe “user stories” around different intended use cases
- Help enforce a user-centric developer mindset
- Provide a minimum viable set of formats and scenarios to support
- Define a set of test cases that need to be checked
- Be as specific as possible. If a use case doesn’t apply to a given story, it may need its own story.

Storyboards: examples

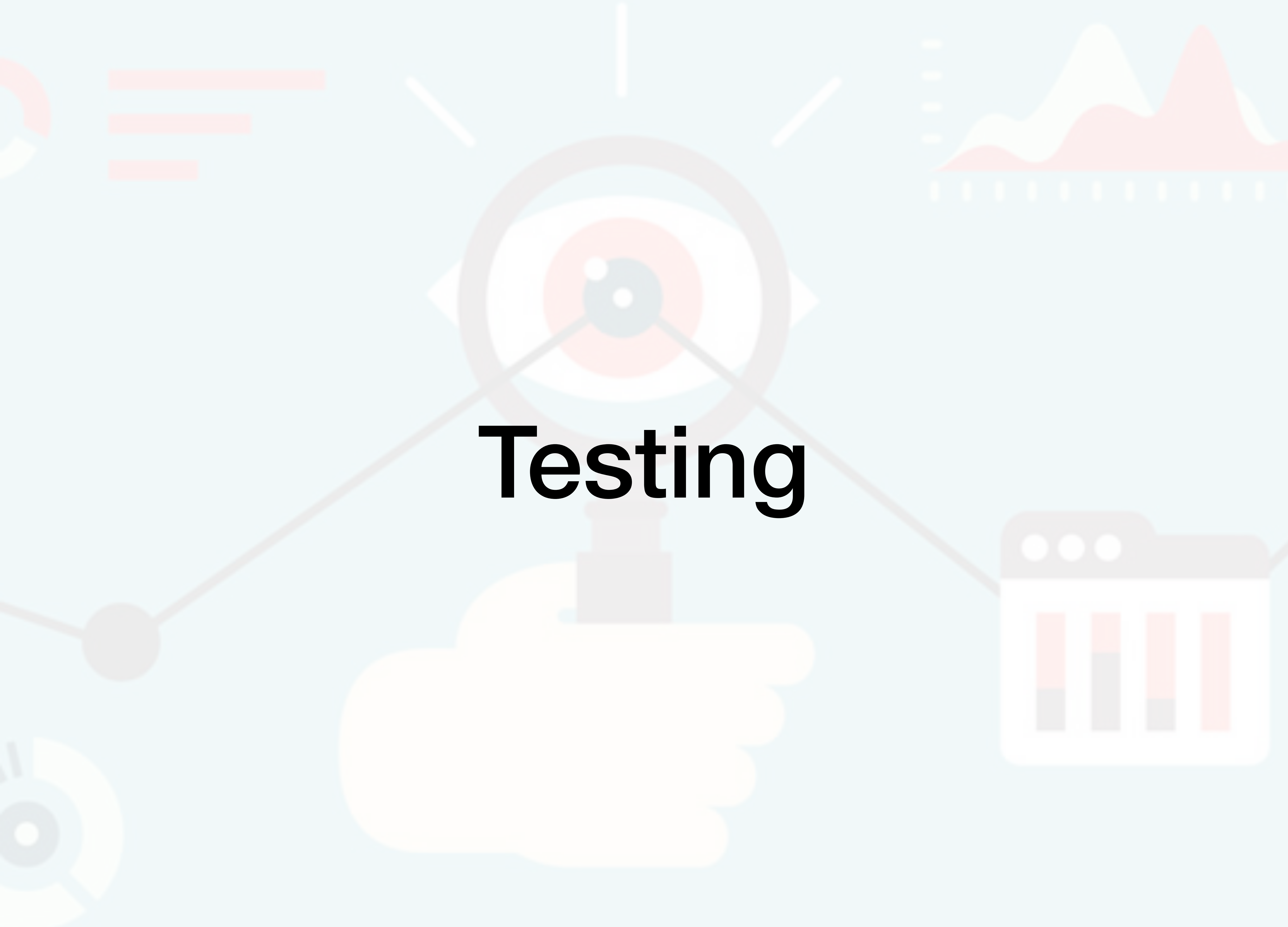
- Alice is a neuroscientist with a collection of structural MRI images. She wants to create detailed images to help her visually identify potential anatomical anomalies in her patients' brains.

- Bob is a psychologist with a collection of functional MRI images. He wants to make animations of brain activity changing over time during different experimental conditions so that he can add a slide to his Keynote presentation.

- Carol is a computer scientist who wants to apply pattern classifiers to structural and functional data from Neurovault. She wants to create a summary plot of which brain regions were most informative.

- Dave is a research assistant who wants to process functional MRI data in near real time as part of a neurofeedback experiment. He needs to read the data, preprocess the images, and predict the participant's mental state within a 2 second window.

Testing



What needs to be tested?

- Test each storyboard and recommended use case
- As new storyboards are added, new tests are needed
- Set up testing **early** to make your life easier in the long run— it’s much easier to start simple and add/modify than to do everything at the end
- Generate a variety of sample datasets and scripts that push on each place that your code might break.
- TravisCI (next week!)

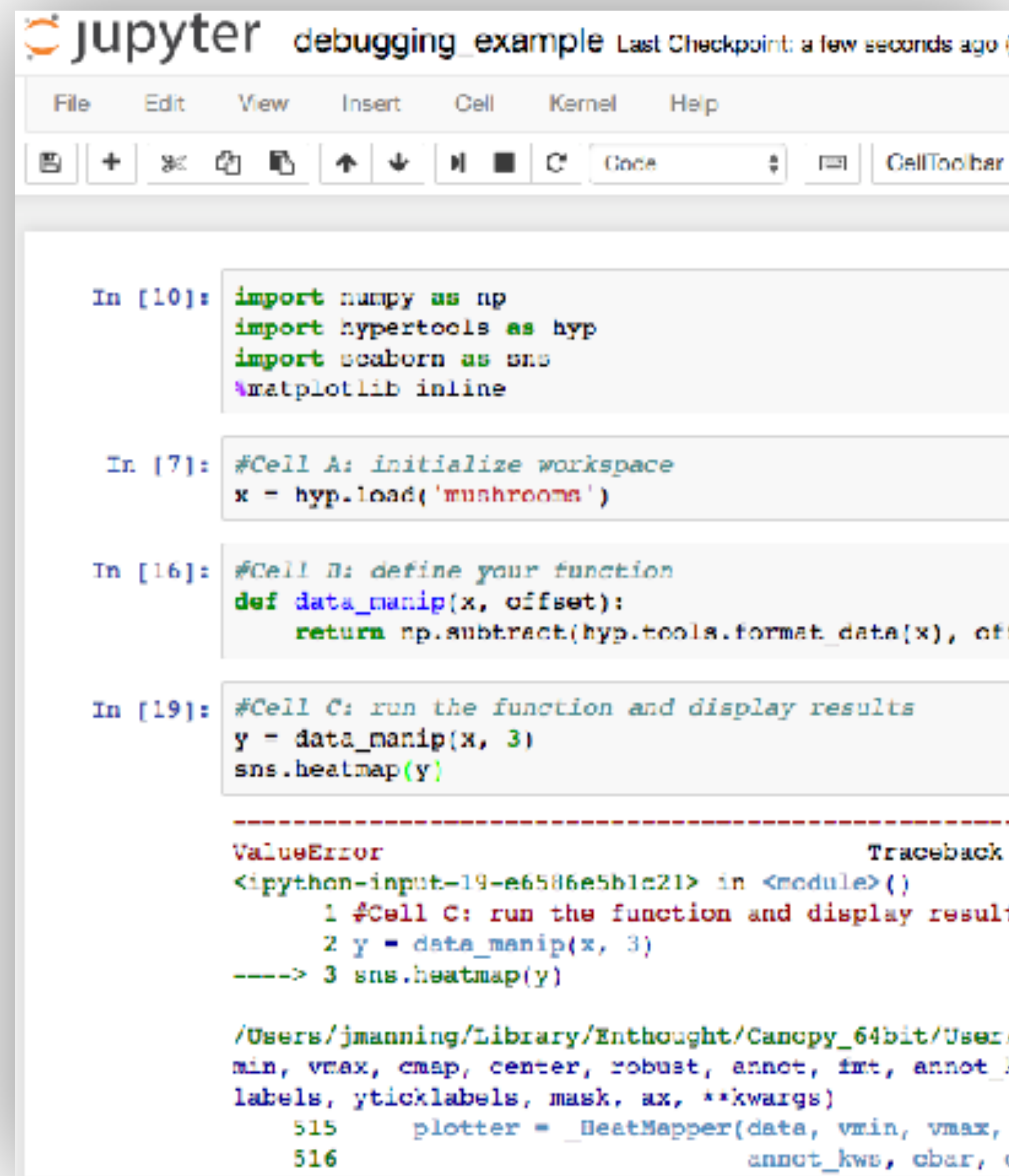
Debugging

Debugging: tips and tricks

- **Section out the one thing you're working on and make it easy and fast to run**
- Use small test datasets to allow you to run your tests quickly
- Minimize the amount you need to do to re-run your test case (e.g. re-typing, re-starting, etc.)
- Ideally identify the simplest scenario where your bug shows up, and just try to fix that (this is sometimes hard)

Debugging: Jupyter

- Initialize your workspace in cell A
- Define your function in cell B
- Run the function and display results in cell C
- To debug, run A, then B, then C



Debugging: PyCharm

- Full tutorial: https://www.jetbrains.com/help/pycharm/part-1-debugging-python-code.html
- Set up Python environment and packages
- Set up breakpoints
- Add a “debug” script for your file
- Use console to interact with (and fix) your code as it’s running

```python
    def demo(self, a, b, c):
        d = b ** 2 - 4 * a * c
        if d > 0:
            disc = math.sqrt(d)
            root1 = (-b + disc) / (2 * a)
            root2 = (-b - disc) / (2 * a)
            return root1, root2
        elif d == 0:
            return -b / (2 * a)
        else:
            return "This equation has no roots"


if __name__ == '__main__':
    solver = Solver()

    while True:
        a = int(input("a: "))
        b = int(input("b: "))
        c = int(input("c: "))
        result = solver.demo(a, b, c)
        print(result)
```

Optimization

When should you optimize?

- Function that will run many times (e.g. “work horse” functions). Time savings are multiplicative with each run.
- Single-use functions with time requirements (e.g. real-time control).

How do you optimize?

- Algorithmically: think through each possible workflow and try to minimize the number of steps that run. **Reduce redundancies** by eliminating repeated steps, pre-computing re-used values, etc.
- Go **back to basics** (as specifically as possible): try to think about how exactly your code is being executed, how data gets read/written, etc. Try to find the slowest steps.
- Focus on **bottlenecks**: identify the slowest part of your code, then move on to the next slowest, etc. until it is "fast enough"
- **Check if a solution already exists**— other libraries, Google/Stack exchange, CS textbooks, code recipes, etc.
- Can your code be **parallelized**? (Multithreading, multiprocessing)

Profiling your code

- In Jupyter notebooks, use `timeit` to analyze the run time of a single cell. This can help identify what's slow.

- The PyCharm profiler is good for more detailed reports:
  https://www.jetbrains.com/help/pycharm/optimizing-your-code-using-profilers.html

  - Set up is similar to creating debugging scripts

Vectorizing

- Loops can often be translated into matrix multiplications
- There is an art to doing this well
- Get to know `lambda` functions and `map`
- Look for existing solutions to similar problems

Vectorizing: example

```
result = np.zeros(x.shape[0])

for i in range(x.shape[0]):

  results[i] = my_function(x[i])
```

Vectorizing: example

```
result = np.array(list(map(my_function, x)))
```

Releasing your code

Your code is ready to be shared when…

- Your code is on GitHub
- Your project well-organized and the syntax is clean
- You have set up automated tests
- You have documented the intended use cases, selected a license, etc.
- Your co-authors approve release

Selecting a license

- When in doubt, choose the MIT License
- If you want to get into the nitty gritties, start here: https://choosealicense.com/

Documentation

- Documentation is about empathizing with your users and future developers (including you)
- Think about the storyboards and intended use cases
- Consider a “gallery of use cases” centered around your storyboards
- Write clearly, explain fully (but simply), cite resources that provide more information (including credit for other people’s ideas)
- Include guidelines for contributing and citing your work
- Sphinx, Readthedocs (another week)

Publicizing your release

- Announce via twitter!
- Make use of the classic science communication tools: posters, papers, talks, etc.
- Consider writing a blog post (e.g. Kaggle) or web-friendly demo (e.g. Distill)
- Organize a hackathon or tutorial
- Word of mouth

Summary

Take-home messages

- **Simplicity** above all else
- Use storyboards to
  - Organize your thoughts
  - Consider theory of mind of the *user*
  - Consider theory of mind of the *developer*
- Funnel early, fail early, and keep the design modular