Coding like a rock star:

Tips and tricks for achieving your science dreams at MIND

Jeremy R. Manning (jeremy@dartmouth.edu)
MIND Summer School 2019
Dartmouth College

Slack: #tutorials

Gettin’ it done

The Zen of Python

>>> import this

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Quick-start guide

- Start with a **GitHub repo**, commit/push **frequently**!
- Find solutions that **work**, even if they're **messy** or otherwise sub-optimal. Take notes (in-line comments or GitHub issues) to remind yourself of things to fix
- Copy liberally from existing code. (Remember to cite your sources!)

Quick-start guide

- Use a **Docker container** to create a stable development environment.
- Use a **Jupyter notebook** to start prototyping the basic functions. Import os, numpy, pandas, matplotlib, and seaborn in your first cell.
- Start with a **“toy” dataset** that will allow you to start attacking the problem. Later you can run your code on a bigger dataset and fix as needed.
- Make **plots** early and often.
- Don’t worry about optimization— first **do things the simple** way that you know works, and **save your intermediate results** (npy, npz, or pkl files) so that you don’t have to keep re-running the same long computations again.

Simplicity

Simplicity

- **Simplicity**: the art of maximizing the amount of work *not* done
- Simplicity means:
  - **Start with the easy stuff** that you already know how to solve
  - Limit the scope of your project to the **minimum viable product**
  - Write each function **once** and reuse it often
  - When in doubt, break the thing your stuck on into **smaller pieces**

Simplicity example: Modular programming

- Idea: design code around *modules* that accomplish simple, general purpose tasks.

- Combine modules to accomplish more complex tasks.

- High-level functions should comprise (only) the main algorithm, with calls to lower-level modules to do anything complicated.

- Goal: someone reading your code should be able to quickly understand what your algorithm is, even for high-level functions.

- Modular programming helps minimize redundant code by facilitating re-use. It also facilitates optimization: optimizing one lower-level model will speed up all higher-level modules that depend on it!

Modular programming

- When you have multiple modules that do similar things, consider:
  - Can I create a new lower-level module that my existing modules could call?
  - Could I create a more general-purpose module and consolidate my code?

When to simplify

- If you are going to be using your code a lot, it's worth simplifying and cleaning it up.

- If you are going to be sharing your code, it's worth simplifying and cleaning it up.

- If you plan to re-visit your code later, or if you want other people to understand your code, it needs to be simple and clean.

- If you are writing a simple "one off" script that you are only going to use rarely (and especially if it isn't going to be shared), a quick and dirty solution is probably fine. Simple sometimes means: get something out quickly and easily.

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 styles

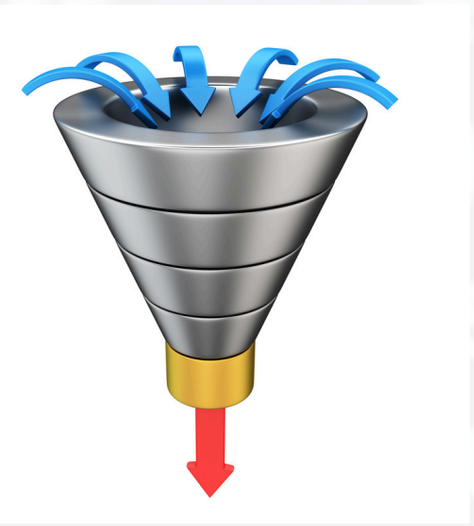
- `x` (single lowercase letter): loop iterators, minor scalar variables
- `X` (single uppercase letter): constants, matrices
- `lowercase`, `lowercase_with_underscores`: variables
- `UPPERCASE`, `UPPERCASE_WITH_UNDERSCORES`: usually constants
- `CamelCase`: classes
- `mixedCase`, `Capitalized_Words_With_Underscores`: 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 your code, “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

When should you funnel?

- If you check for formatting-related properties (e.g. class types, data dimensions, presence of particular arguments), this should be done **once** at the beginning of your function (and only in functions that the user interacts with directly).

- If you have a lot of “special cases” in your code, you may be able to simplify by funneling.

- Internal functions, or functions that only support one use case, probably come “pre-funneled.”

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. Avoid "mirrored" structure across different folders, unless that is a specific design feature (e.g. data format, tutorial format in this repo).

Storyboarding

TITLE SCREEN
FADE IN FROM BLACK
6 seconds

SPACE SHIP HOVERS FOR A MOMENT AND THEN FLYS TOWARDS A DISTANE EARTH
5 seconds

SPACE SHIP ON SURFACE OF MARS
4 seconds

SPACE SHIP FLYS OVER CITYSCAPE
5 seconds

ALIEN ENTERS INTO SPACE SHIP
4 seconds

PERSON ON GROUND SPOTS SPACE SHIP
6 seconds

Storyboards

- Describe “user stories” about different intended use cases. Try to imagine **why** the user is here and **what** they are trying to do.

- Help enforce a user-centric developer mindset.

- Provide a minimum viable set of formats and scenarios to support. This defines the project scope.

- 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...or it may be beyond the intended scope of the project.

Storyboards: examples

- Alice is a neurologist 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

Do you need to test your code?

- Yes.

Who is exempt from the “test everything” principle?

- No one.

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.

- PyTest and TravisCI [tutorial]

Debugging

Debugging: tips and tricks

- **Section out the one thing you're trying to fix and make it easy and fast to run (i.e., "get to that point" in the execution pipeline).**

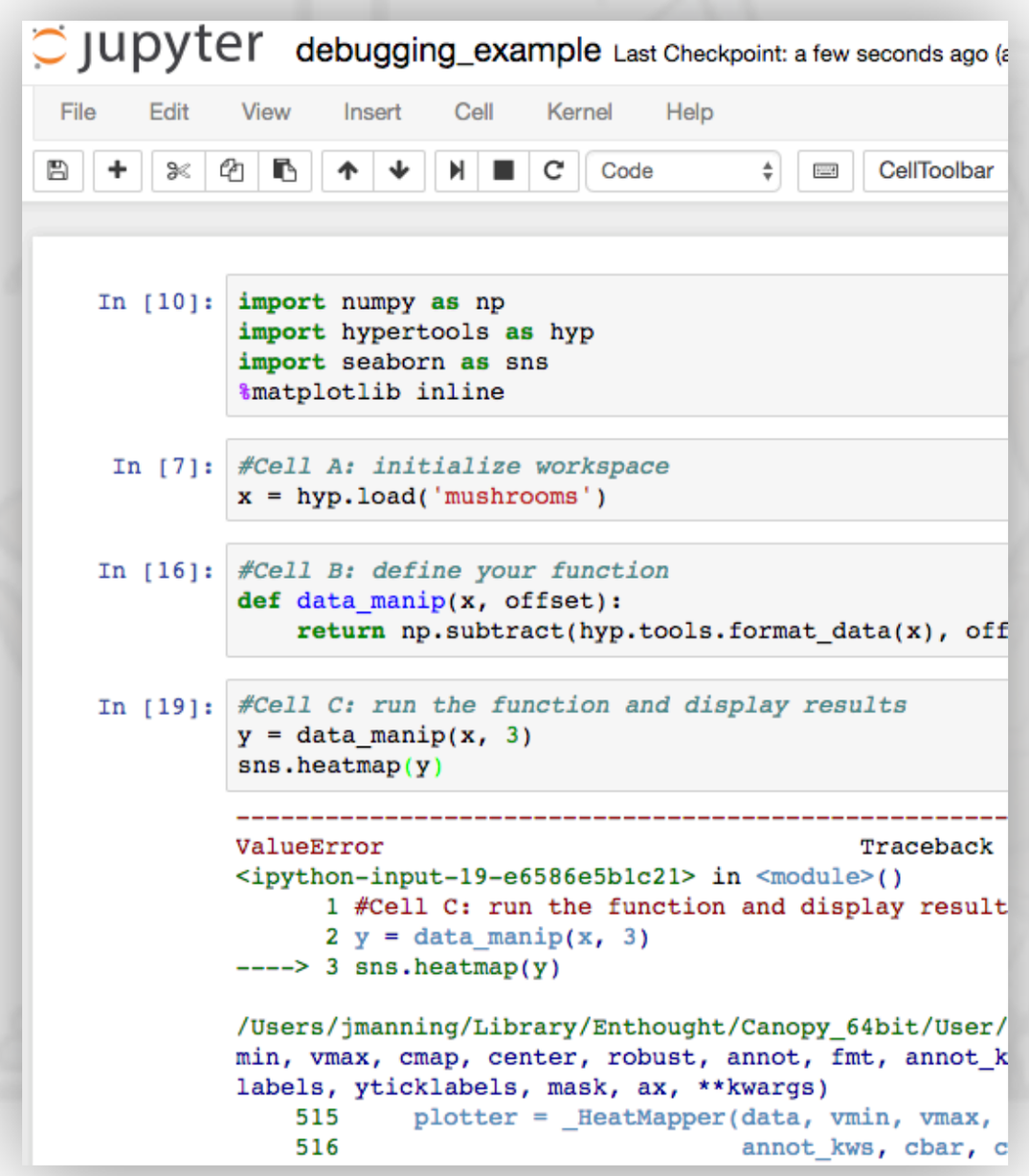
- Use small test datasets to allow you to run your tests quickly.

- Minimize the amount you need to do to re-run your test case (e.g. re-typing, re-starting, etc.). For a tricky debugging session you might be running the same code hundreds of times. Make this easy!

- Ideally identify the simplest scenario where your bug shows up, and just try to fix that (this is sometimes tricky).

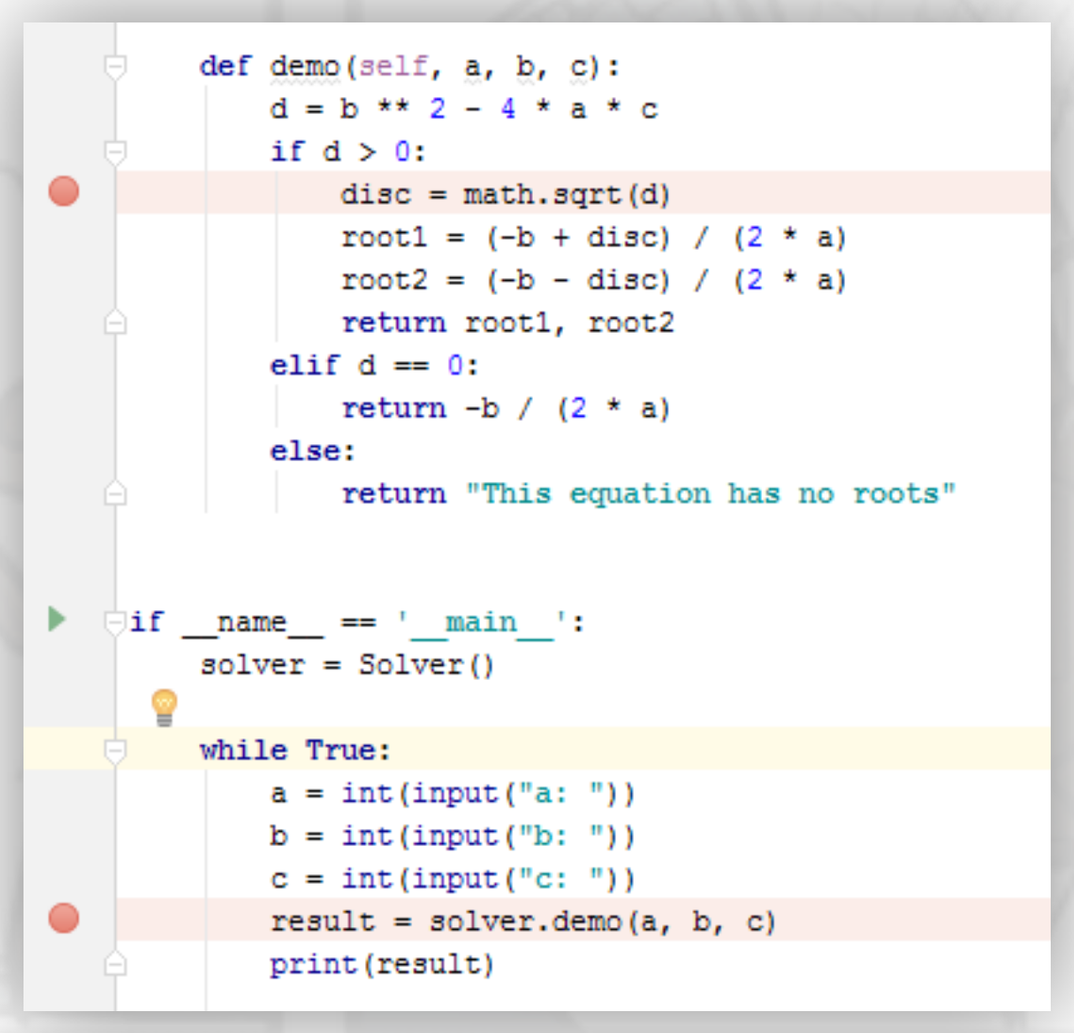
Debugging: Jupyter

- Good for rapid debugging
- Basic setup:
  - Initialize/reset your workspace in cell A
  - Define your function in cell B (this is what you'll be modifying)
  - Run the function and display results in cell C
  - To debug, run A, then B, then C



Debugging: PyCharm

- Good for complex bugs
- Full tutorial: [link]
- 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



Take-home messages

- Your goal this week is to create **minimum viable products** as quickly as possible. (Later you can clean them up and optimize.)
- **Simplicity** above all else. Something is better than nothing.
- Use **storyboards** to organize your thoughts
- **Funnel** and keep the design **modular**
- **Ask for help** and **learn** from others! (Corollary: help people when they ask!)