theory

adam okulicz-kozaryn

adam.okulicz.kozaryn@gmail.com

this version: Wednesday 16th April, 2025 10:45

outline

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

[*] other thoughts

Know Your Data!

- simply cant use it well if you dont know it well
  - (not just data; the field: theory, lit, method, etc)
  - this is where you beat IT folks (MS/PhD just in IT)

    https://www.google.com/search?q=data+science+venn+diagram
- again, be prepared to invest a lot of time into your data
  - use data that you're passionate about
  - or that can make $ (now or in future career)
- don't forget to think about it! don't be mindless!
  - ask questions, be investigative, be critical
- double check, cross check, give to others to check

outline

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

[*] other thoughts

Wilson put it well

- print out Box 1 from these 2 art
  - hang it at your office, home, and elsewhere
- `https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745`
- `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510`
- and Know Your Data (2nd slide)
- and variations on these, and other general rules follow

simplicity, cleanliness, and organization!

- keep it as simple as possible
  - especially if overwhelmed or struggling
- say retain only 5vars and 25obs
  - much easier to understand such data
- simplicity transpancy clarity:
  - use fancy code: eg loops iff they simplify
- have chunks of code only once
- code it all from raw to final (replication principle)
- organize: sections, comments, and logical order (eg rewrite, move code around)

be fast/efficient

- the fancier the code, the more time/effort to write it
- don't do fancy things unless they save time in the long run
- it's all about managing complexity
- automate as much as you can
- simplify and be clear
- have general modules (sections or separate files)
  - that can be reused for different projects
- be lazy: don't reinvent the wheel–google often

outline

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

[*] other thoughts

accuracy / correctness

- ◇ it's fundamental and obvious: bad if wrong
- ○ double check
- ○ especially at the beginning do not assume things
- ○ double/triple check the whole code once finished
- ○ go public, present, give it to others
- ● its human to err, there will be mistakes

1.write; 2.rewrite/reorganize; 3.optimize/improve

- 1. dump it, do "free writing" with code, too (i often come up with some idea out of sudden, and just write it down)
  - start simple and keep on adding things
- 2. rewrite/reorganize your code
- 3. optimize/improve
  - dont optimize too early, first make it work
  - (tendency to over-optimize! eg spending days on small chunk of code that does not really matter)
- (then can rewrite/reorganize again)

rewrite and reorganize

- efficiency: few lines of code do many things
  - do more in fewer lines, drop unnecessary things
- reorganize and rewrite!
  - just like your papers: you print them out
  - and move paragraphs and words around
  - and you simplify and strike out unnecessary words
- do the same with code! drop everything you can!
  - code should be "tight"
  - as few lines as possible to perform given task
- use wrappers on more laborsome functions, google it, stack-overflow, etc
  - instead of lists pandas, instead of mpl sns, etc

efficiency: on the other hand

- but you also want to be extensive in a way
- in a good way...
- like with free writing, so with code
  - do “free writing”
- be expressive and dump your ideas into notebook
- just be organized so that you know what is going on!
- yes, by all means, be efficient–drop unnecessary things
- but do not drop things that may be useful
  - say in the future or other projects
  - may comment them out, move to LATER sec, etc

optimize/improve/get fancy

- think how to optimize
- related to efficiency: do more in fewer lines
  - but here instead of dropping unnecessary stuff
  - we get fancy: loop, list comprehension, your own function
- but first: dont reinvent the wheel, search for existing functions
  - only then optimize yourself, eg loop, write your own function

standardize

- ◇ fewer mistakes (fewer decisions, a template, on autopilot)
- ○ transparent code , easier to find weird stuff, errors
- like have a template for some vis: say always hist for all key vars; scatterplot matrix for all continuous vars, etc; like a profiler
- and then have the creative part, vis for specific project

modularity

- break large tasks into small (manageable) blocks/components
  - (like in dissertation–don't overwhelm yourself doing everything at once)
- the components are like sections in a paper, step-by-step
- it is easy then to reuse these components
- so have separate ipynb for different tasks; and have sec and subsec within ipynb

automation (closely related to standardization)

- everything should be coded
- no copy-paste, point-and-click, etc
- dont use excel for anything!
  - ◇ automate as much as possible!
  - ◇ practical reason: faster! (in the long run)
  - ◇ technical reason: computers *never* make mistakes
  - ◇ eg pull automatically from database, upload to github
  - ◇ btw human automation is great too:

    `https://www.amazon.com/`

    `Power-Habit-What-Life-Business/dp/081298160X`

document

- have text fields in ipynb and # comments in code fields
- meaningful commit messages in git!
  - go thru a student github commit history
  - commit messages do help to document
  - and show yet again diff in colab on student's progress
- may have changelog (version, date and explain what changed, eg: 0.1 dumped raw ideas; 0.2 loaded X data; 0.21 loaded Y, Z data)
- difficult to overestimate importance of documentation
- note: typically, i underdocument, too

singularity

- have only one chunk of code and one file in one place, ideally in git
- as projects grow, get complex and maybe branch off, do have parents-children, and possibly branches
- elaborate: eg i do a lot with gss, draw flow-chart
- at first it was a first paper
- then i write 2nd paper, and realize most of dat man is the same
- so i create one root/parent with common code for all gss papers

portability

- import libs at the beginning, the fewer the better
  - i overdid, i need to cut down
- can get version of key libs, ie pandas and save info
- but here few libs, easily trace back if it breaks in few years
- do always save raw data! in couple years dataset may disappear or change
- magics (eg %matplotlib) for running in different environments eg spyder

outline

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

[*] other thoughts

more principles

- 2 books about general programming (classics and free!)
  - `http://catb.org/esr/writings/taoup/`
  - `http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-1.html`

clarity

- "design for transparency and discoverability"
  - write clean code [eg split 1 fn over many l for readability]
  - avoid fancy code
  - fancy code is buggier
  - clarity is better than cleverness
- eg:
  - group logical chunks together
  - more than twice nested loops gets confusing
  - if your code is mostly loops, consider functions

modularity

- "write simple parts that are cleanly connected"
- "controlling complexity: the essence of programming"
  - debugging dominates development
- eg:
  - better many small loops that each does one thing, than one huge (>100 lines) loop that does everything
- organized code logically not chronologically
  - do free writing, but then reorganize
  - like with papers, code should be rewritten, eg:
  - no data management in data vis part

composition

- “design programs to be connected to other programs”
- eg clean data in one sec to make it ready for another sec
- the workflow needs to be logically organized

optimization (fancier, fewer lines)

- yes, but "get it working before optimizing" !
- eg:
  - first make mpl hist for one var, make it working
  - and then deploy it for 10 vars with a loop

extensibility

- "design for the future because it will be sooner than you think"
  - you will reuse your code in the near future
  - so write it clean
  - have sections, etc
  - use lots of comments
  - reorganize, rewrite
  - optimize

silence

- "when a program has nothing surprising to say, it should say nothing"
- drop unnecessary code
  - if you think it may be useful in the future comment it out
- do not generate unnecessary output, do not lose your reader in unnecessary clutter
  - if the output has nothing useful to say it should be dropped
  - (or commented out)

automation (again)

- "rule of generation: avoid hand-hacking"
- because humans make mistakes and computers don't, computers should replace humans wherever possible
- automate anything that you can
- but stay human, focus on fun creative part, eg vis
- dont automate everything; eg not all vis mindlessly

save time: reuse, cp-paste, dont reinvent the wheel

- if someone already solved problem, reuse it!
- very unlikely you are doing something completely new
  - eg google 'student data analysis python'
- if anything, the problem is that people do not share their code
- usually all need to do is to adjust somebody else's code or your old code
  - its like doing lit rev, but with code
  - and with data too, eg google scholar "psid biking" to find out how people use biking var in psid

save time: reuse, don't reinvent the wheel

- ask people for code:
  - your supervisor
  - journal article authors
  - your colleagues, friends, etc
- share your code
  - you may want to protect some parts of it
  - (critical, innovative research ideas, etc)
  - but share as much as possible
- acknowledge others' work

defensive programing

- "people are dumb-make program bullet-proof"
  - you will find negative income, age over 200
- think of likely possibilities/instances; especially if you suspect some specific problems
- thats also why its so important to interpret critically your vis, if something looks funny or unlikely, there's a mistake

[*] cant go wrong with google recommendations

- these 2 classic books ver classic, like 90s classic
  - as opposed to tools, programs, functions, theory ages like fine wine but still
- here's an uptodate take on Py dos and donts:
- https://google.github.io/styleguide/pyguide.html
- but not much fun, rather technical CS than fun DS, dry but comprehensive

outline

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

[*] other thoughts

- 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!

outline

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

[*] other thoughts

simplicity: different, often opposite, from optimization

- ◇ people don't realize this!
- be as simple as possible in writing the code (papers, too)
- ◇ the more code you have (always try to get less) and the more complicated (optimized) it is:
  - ○ the more likely you have mistakes
  - ○ and the more difficult it is to find them
- do not complicate (optimize) your code for the sake of fanciness
  - ○ yes simpler is better

tradeoffs: life is difficult

- simplicity is sometimes inversely (positively) related to efficiency (amount of code/being verbose) (as you cut stuff down, it may take longer to figure it out (parsimounious is sometimes complicated)
- simplicity usually inveresely related to optimization (eg loops)
- simplicity often inversely related to automation (eg complicated code with if else)

cont

- so make choices, the more serious you are about coding, the more work you do:
  - the more you should care for automation, efficiency, and optimization
  - the more automation/efficiency/optimization actually simplifies
- like Py v excel: excel simpler for simple tasks
  - but Py is simpler for complicated tasks
- stick with one soft! [i think i'm quitting stata]

tradeoffs: life is difficult

- ◇ so make choices, the more serious you are about coding, the more work you do:
- ◦ the more you should care for automation, efficiency, and optimization
- ◦ the more automation/efficiency/optimization actually simplifies
- • like Py v excel: excel simpler for simple tasks
- ◦ but Py is simpler for complicated tasks
- • ideally stick with one soft [maybe i'm quitting stata]