

vis theory

adam okulicz-kozaryn

adam.okulicz.kozaryn@gmail.com

this version: Friday 17th February, 2023 19:34

outline

theory (Tufte)

exemplary examples (see also notebook sec)

graph basics

actual graphing

general programming rules, esp scientific computing
elaboration, details, in other words
and yet one more set of rules
the zen of Python

why vis? why bother? whats the big picture?

- understanding your data is fundamental/critical for research/insight
- vis is the best way to understand data
- data are numbers, usually many and in a matrix
 - vis allows humans to comprehend those many numbers
 - if you look at numbers you will be slower in understanding
- pictures are not less “scientific” than numbers!
- actually the very best journals mostly do graphs!!!
 - nature, science, pnas, psy sci, etc

outline

theory (Tufte)

exemplary examples (see also notebook sec)

graph basics

actual graphing

general programming rules, esp scientific computing
elaboration, details, in other words
and yet one more set of rules
the zen of Python

make it interesting!

- The greatest value of a picture is when it forces us to notice what we never expected to see.

references/links

- Tufte (multiple) <http://www.edwardtufte.com/tufte/>
- Kosslyn “Clear and to The Point”
<http://www.amazon.com/Clear-Point-Psychological-Principles-Presentation/dp/0195320697>
- <https://chart.guide>
- <https://datavizproject.com>

be simple: avoid clutter in general

- everything should be made as simple as possible,
 - but not one bit simpler
- avoid padding: present only data needed for a specific purpose
- avoid clutter: eg single graph must only present the data that are highly related and must be compared
- put data into appendix if it is not very relevant but may be useful
 - people looking for extra information will find it
 - people interested in the main story will not get distracted

avoid visual clutter

- all parts and attr of graph must be meaningful
 incl:
 - shades
 - colors
 - decoration

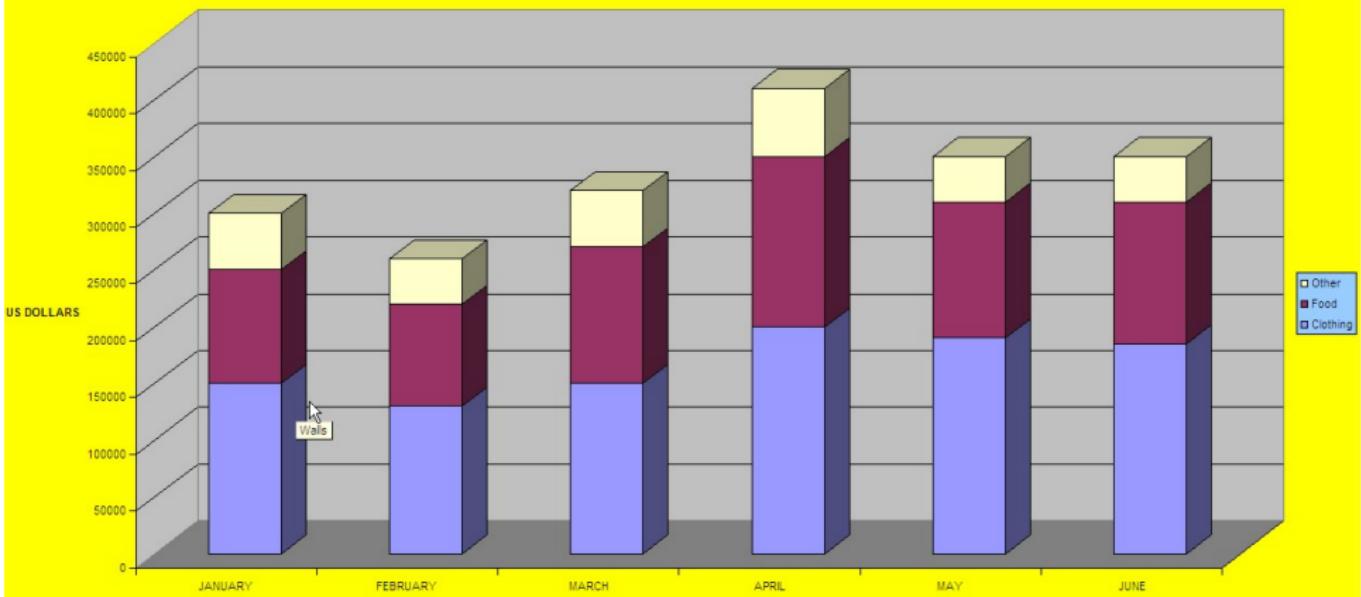
chartjunk

The interior decoration of graphics generates a lot of ink that does not tell the viewer anything new. The purpose of decoration varies to make the graphic appear more scientific and precise, to enliven the display, to give the designer an opportunity to exercise artistic skills. Regardless of its cause, it is all non-data-ink or redundant data-ink, and it is often chartjunk.

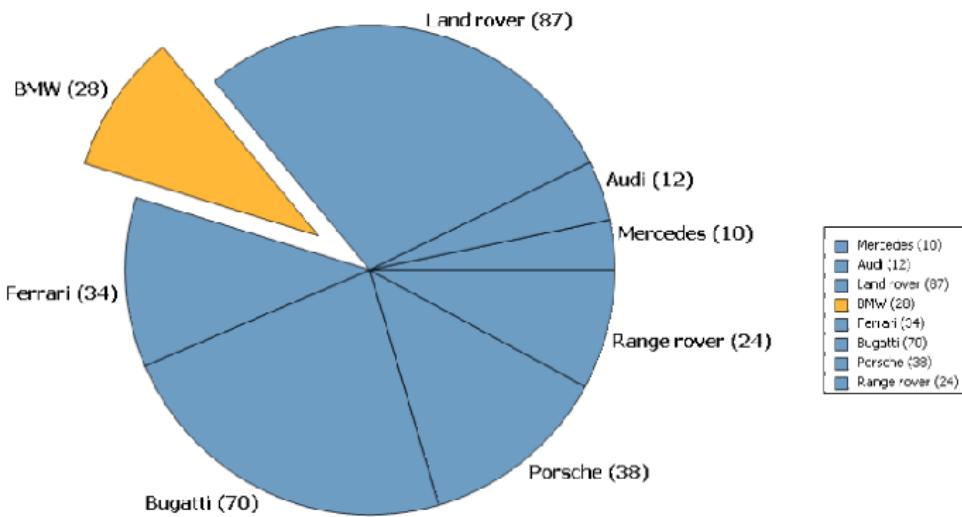
● Edward Tufte “The Visual Display of Quantitative Information”

- examples follow:
 - chartjunk or “business graphs”
 - good fancy graphs

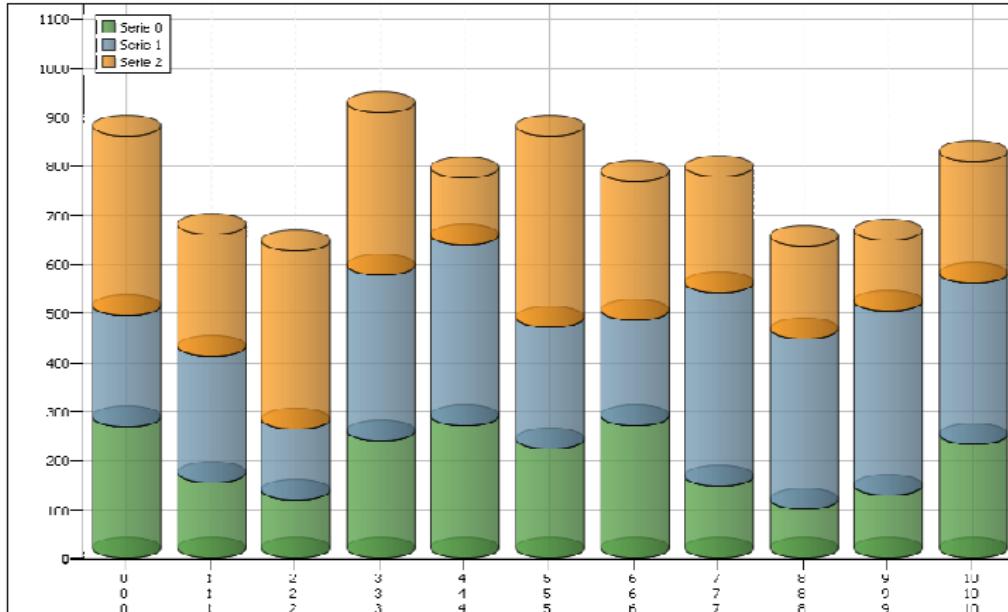
SIX MONTH SALES TOTALS



Exploded Pie Slice Chart



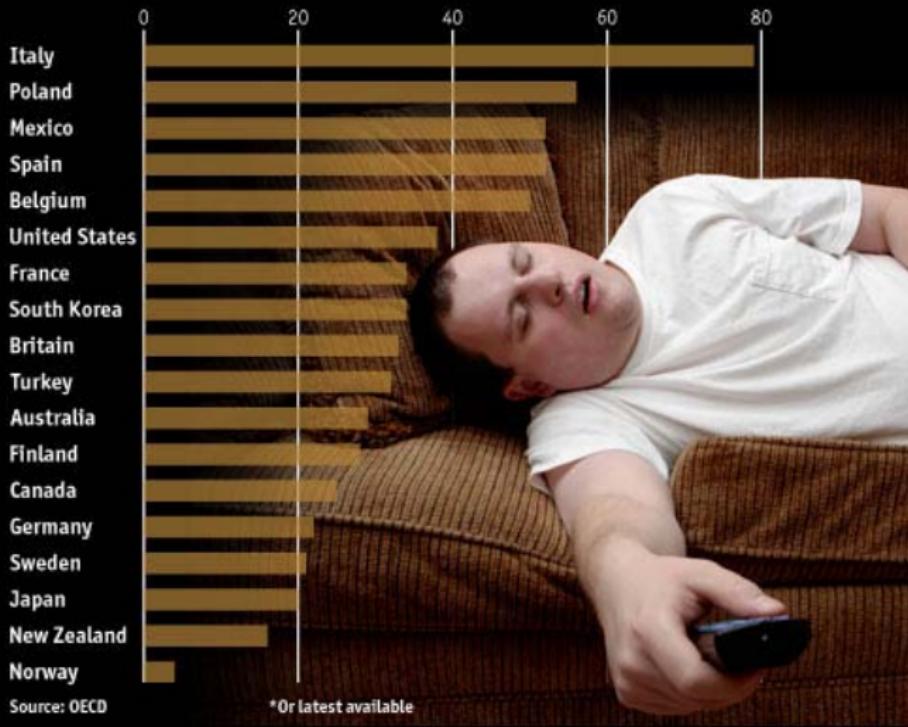
Cylinder Chart



not chartjunk (the economist)

Extra leisure time enjoyed by men compared with women

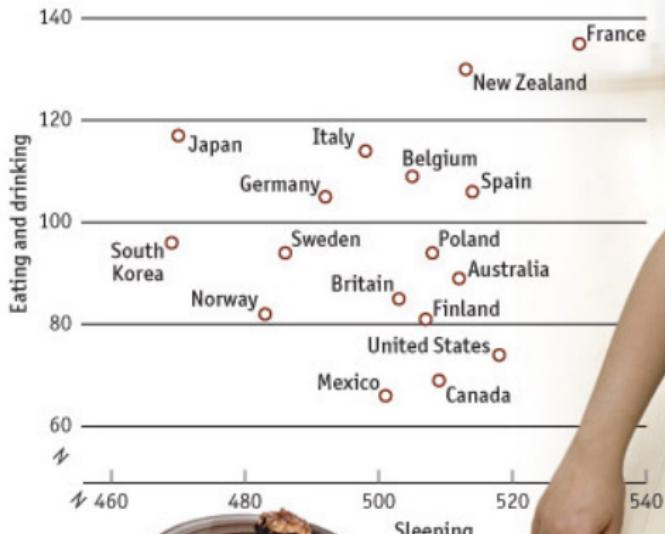
2006*, minutes per day



not chartjunk (the economist)

Time spent eating and sleeping

2006, minutes per day



Source: OECD

balance! keep balance in:

- colors (say either use toned down colors for everything or use stark contrasts, etc)
- fonts: titles, notes, labels etc should be proportional
- thickness of lines
- and everything else
- in general: rather use less ink than more
- note: my classic 90s mpl theme tends to be heavier on ink
 - but then I do make sure to keep it as barebone/simple as possible

one v several vis

- usually to convey an idea, may need several vis
- say if you want to show deprivation in SJ:
 - eg low educ, poverty, crime, etc etc
 - but can also show a summary, eg an index
- sometimes better one vis
 - eg showing change: better one than two; just calc chng var
 - perc chng in $100 * (\text{pop10} - \text{pop00}) / \text{pop00}$

think about it/meaning: the 'so what?' question!!!

- ok, you've got the vis
- now think about it...
- what does it mean? interpret substantively!
 - (beyond technical correctness; lack of mistakes)
 - eg is it interesting, informative, or helpful, etc?
 - as you look at it, ask yourself the 'so what?' question
 - if not happy with the answer:
 - drop it, comment on it, or leave it as it is, AND
 - keep going, and produce a better vis
- in this class I grade substantive meaning, too
- in fact the idea, the meaning, the contribution to the knowledge is most important!

outline

theory (Tufte)

exemplary examples (see also notebook sec)

graph basics

actual graphing

general programming rules, esp scientific computing
elaboration, details, in other words
and yet one more set of rules
the zen of Python

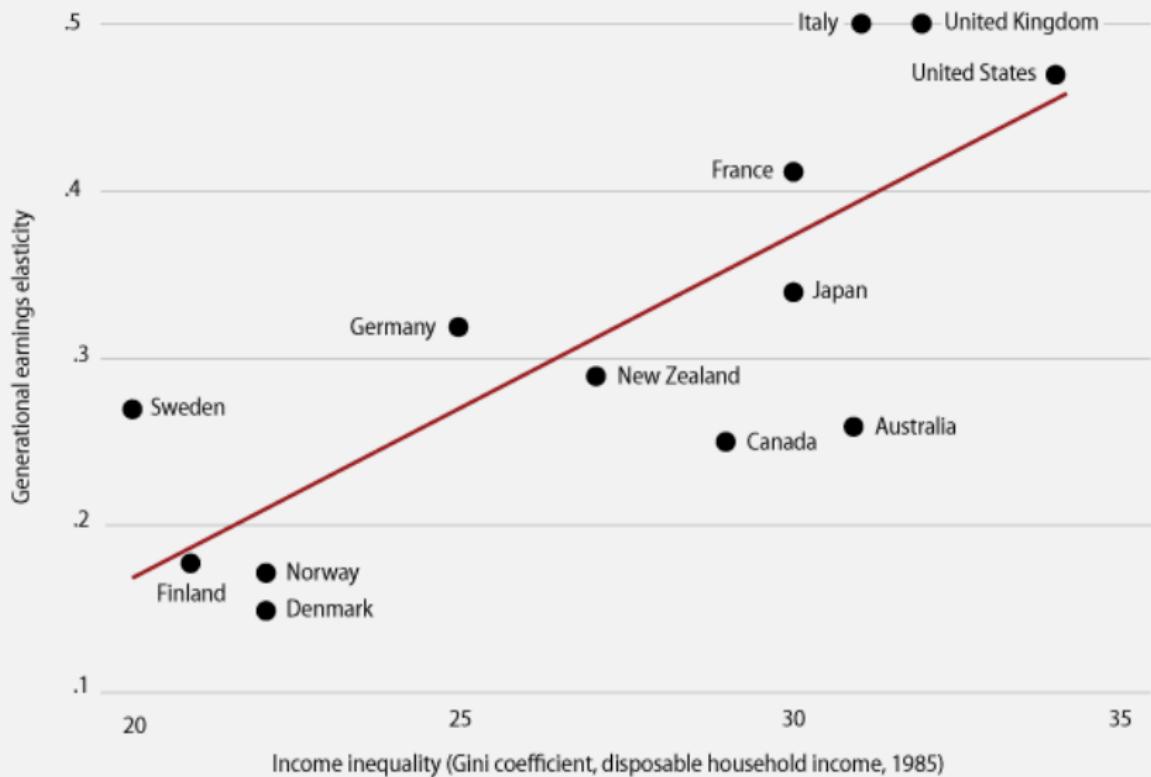
examples

- now that we have covered vis, time for you not just use it correctly,
- but use it to produce new knowledge
- below examples to inspire-vis (usually simple!) producing great insight—what counts most is an idea!
- [to find out more about the vis/research just google vis title]

FIGURE 1

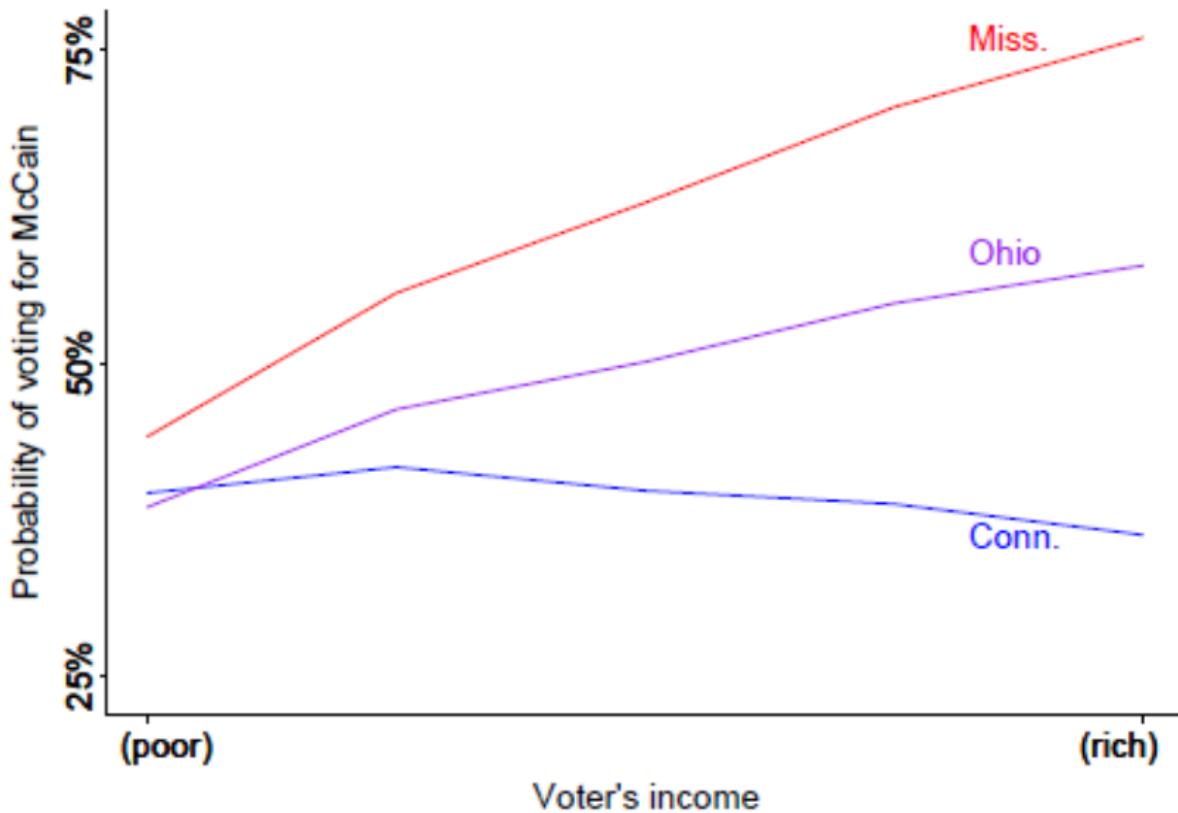
The Great Gatsby Curve

More inequality is associated with less mobility across the generations

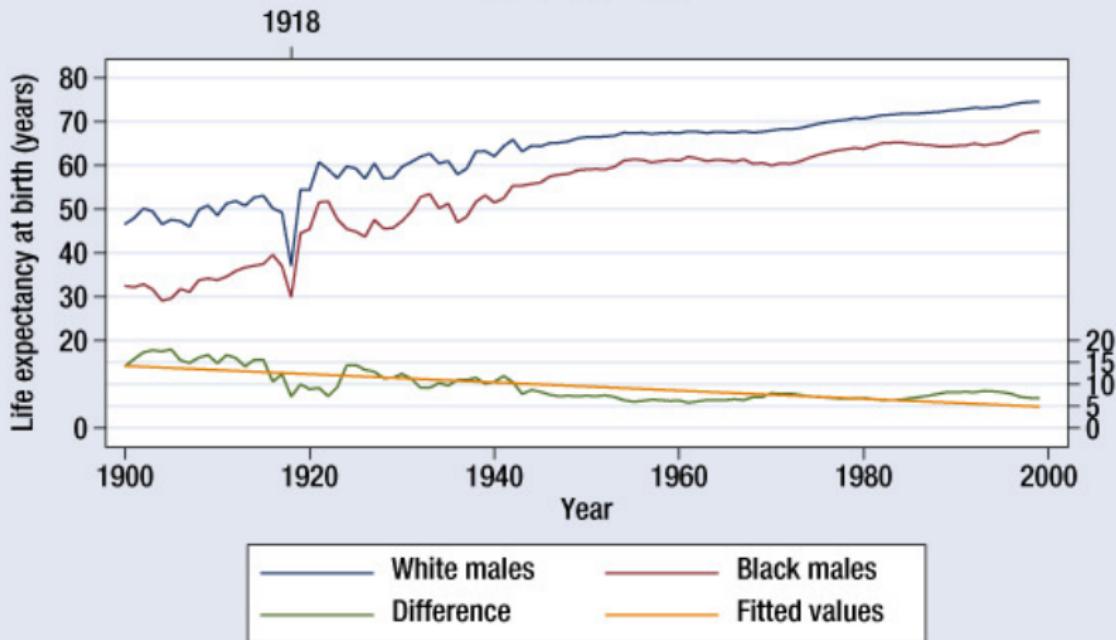


Source: Miles Corak, "Inequality from Generation to Generation: The United States in Comparison." In Robert Rycroft, ed. The Economics of Exemplary Examples (see also notebook see) The 21st Century (Santa Barbara, California: ABC-CLIO, 2013).

McCain vote by income in a poor, middle-income, and rich state

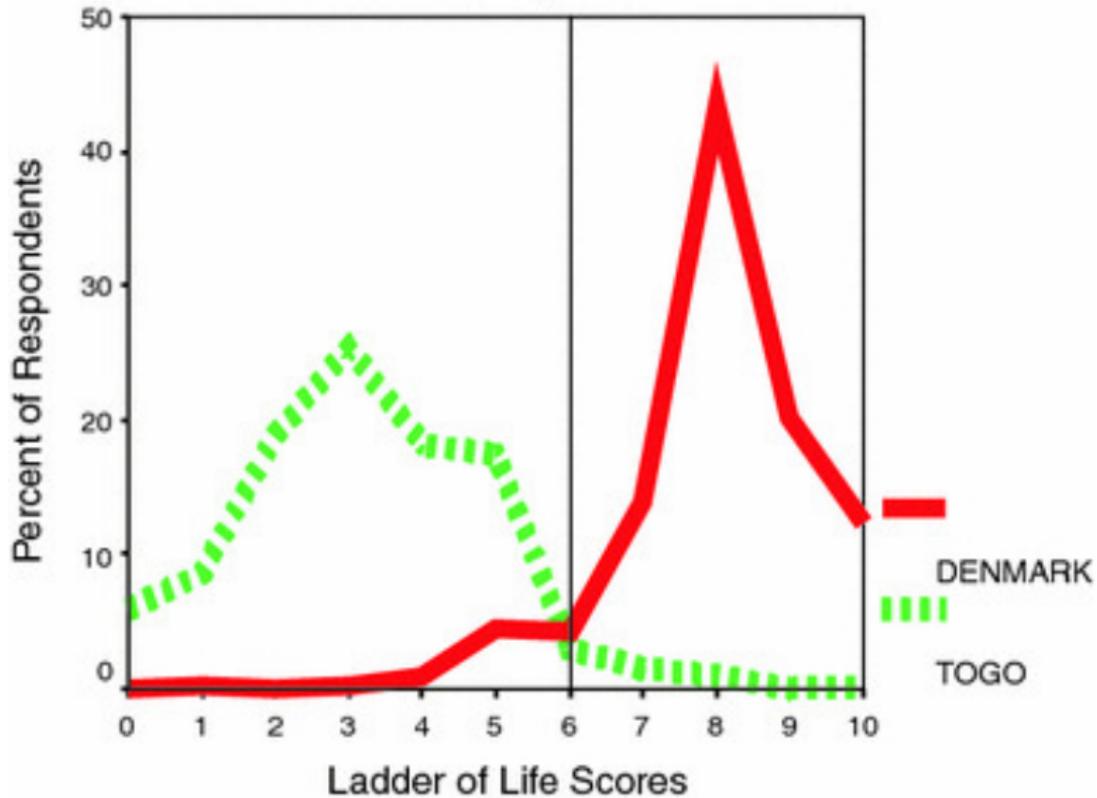


White and black life expectancy USA, 1900–1999

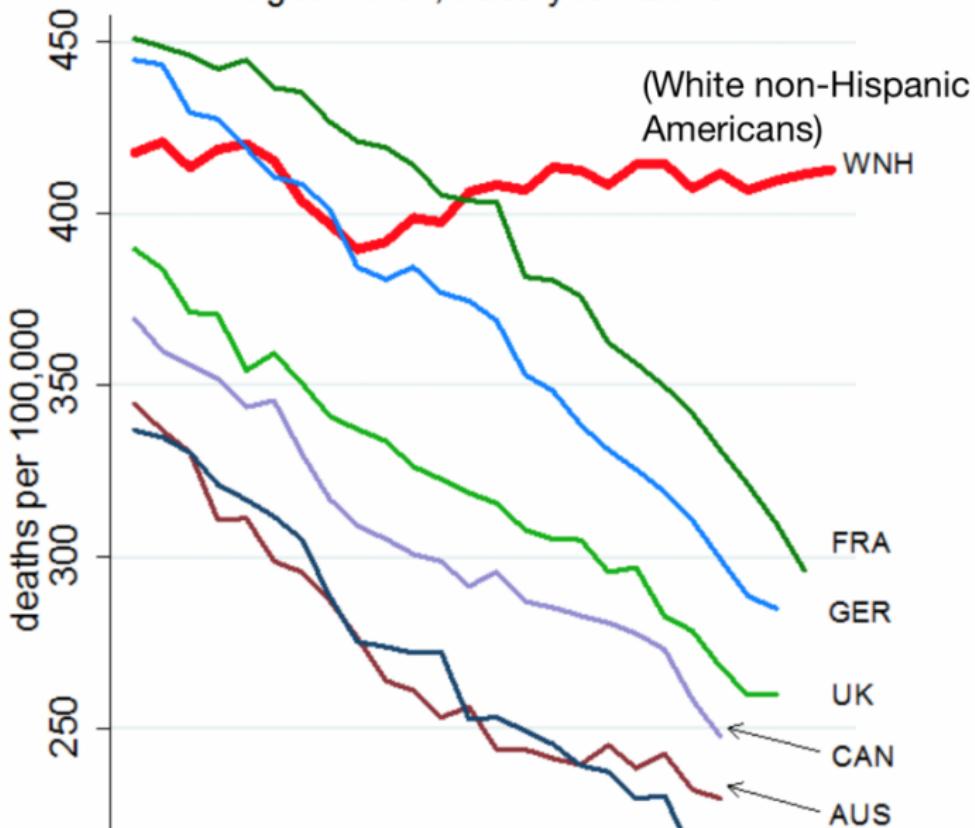


Source: National Vital Statistics, Vol 50, No. 6
(1918 dip caused by 1918 Influenza Pandemic)

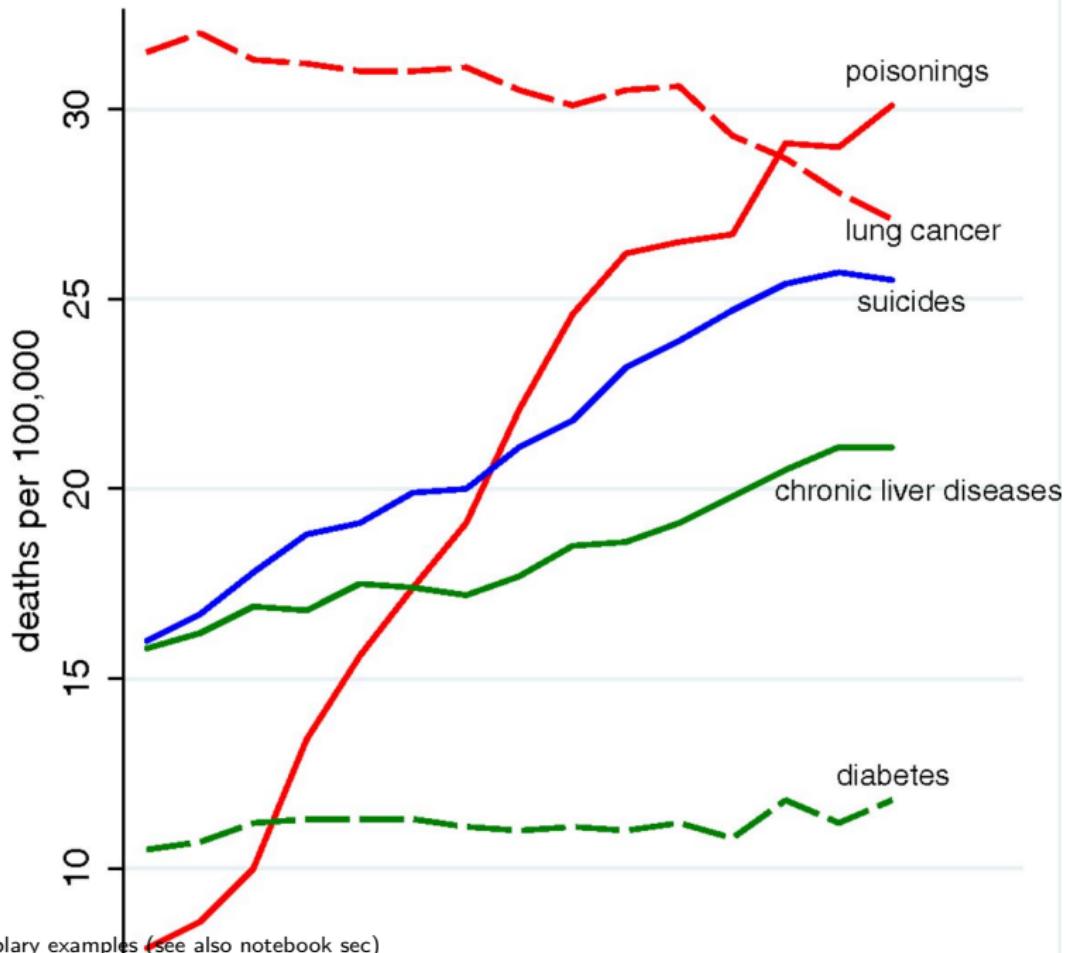
94 % of Danes are Above
97 % of Togolese

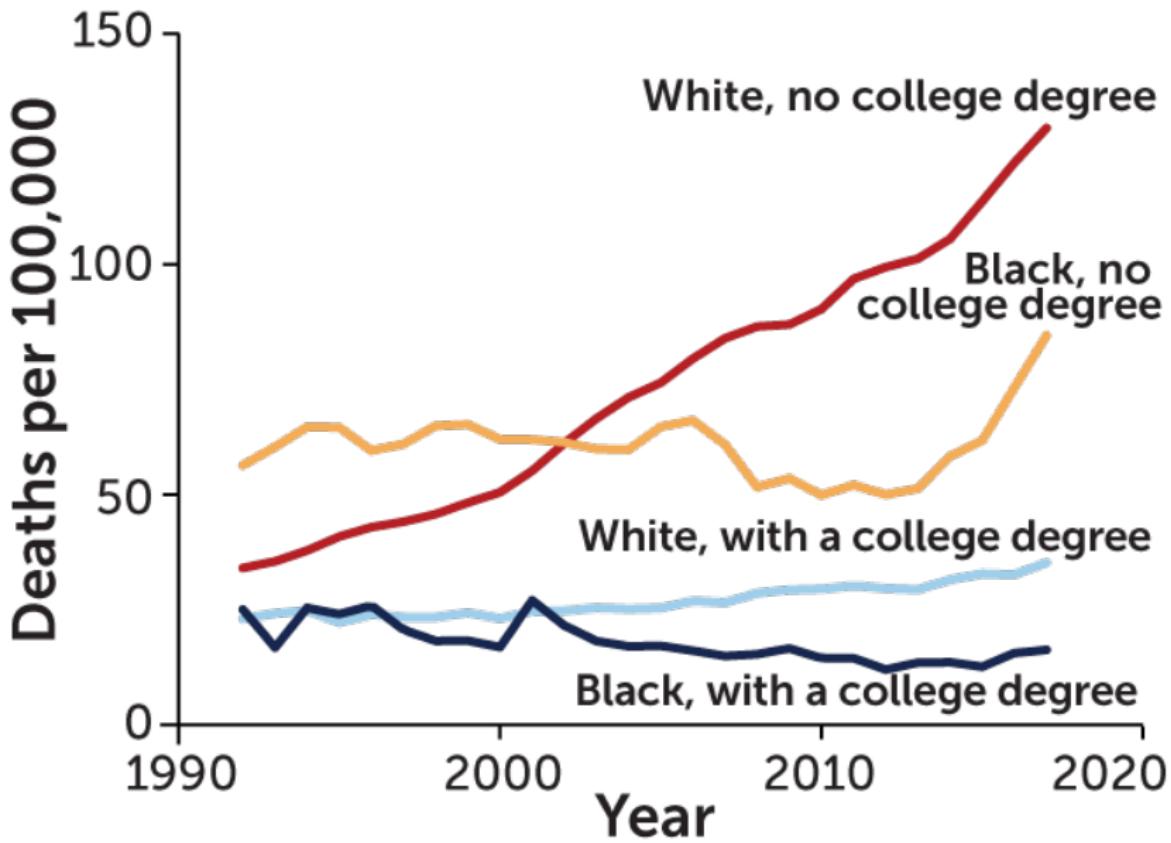


Age-adjusted mortality rates
ages 45-54, base year=2010

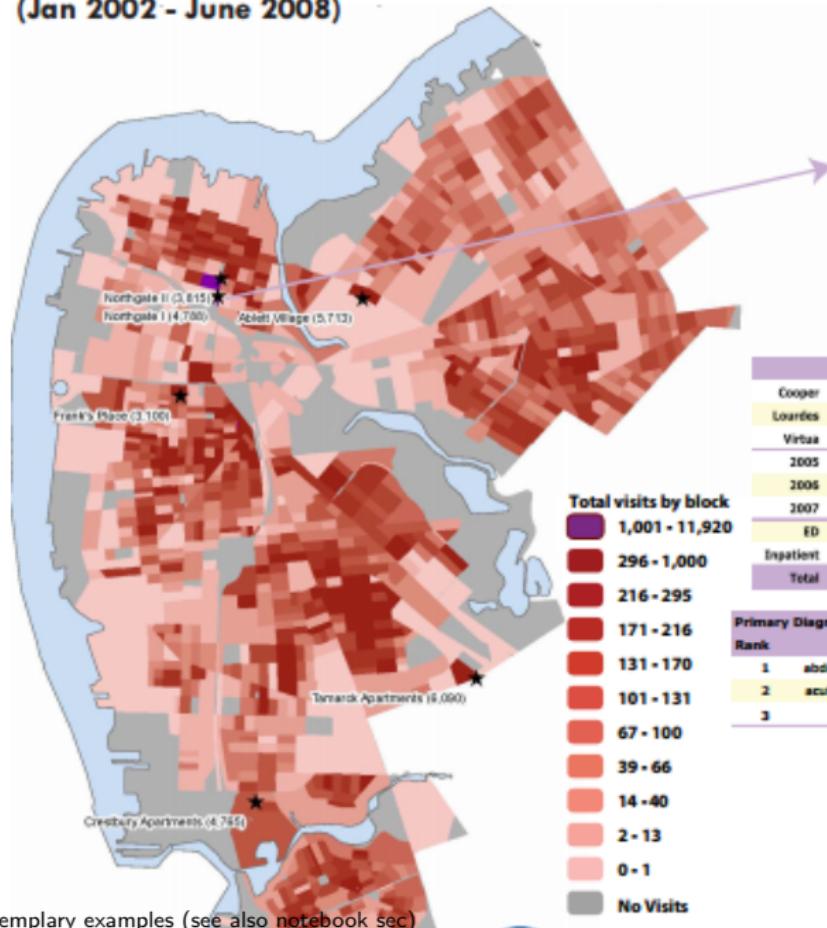


exemplary examples (see also notebook sec)





Inpatient and Emergency Room Visits in Camden, NJ (Jan 2002 - June 2008)



Northgate I Public Housing



	Visits	Patients	Charges	Receipts	Collected
Cooper	3,172	749	\$4,144,897	\$4,994,658	12%
Lourdes	811	337	\$7,848,899	\$1,038,611	13%
Virtua	805	331	\$1,742,487	\$345,092	20%
2005	838	370	\$10,834,420	\$1,288,373	12%
2006	738	355	\$6,867,995	\$883,549	13%
2007	790	369	\$7,979,262	\$961,181	11%
ED	3882	908	\$6,150,592	\$864,019	1%
Inpatient	906	408	\$45,584,781	\$5,504,342	12%
Total	4,788	1,679	\$51,735,374	\$6,388,381	12%

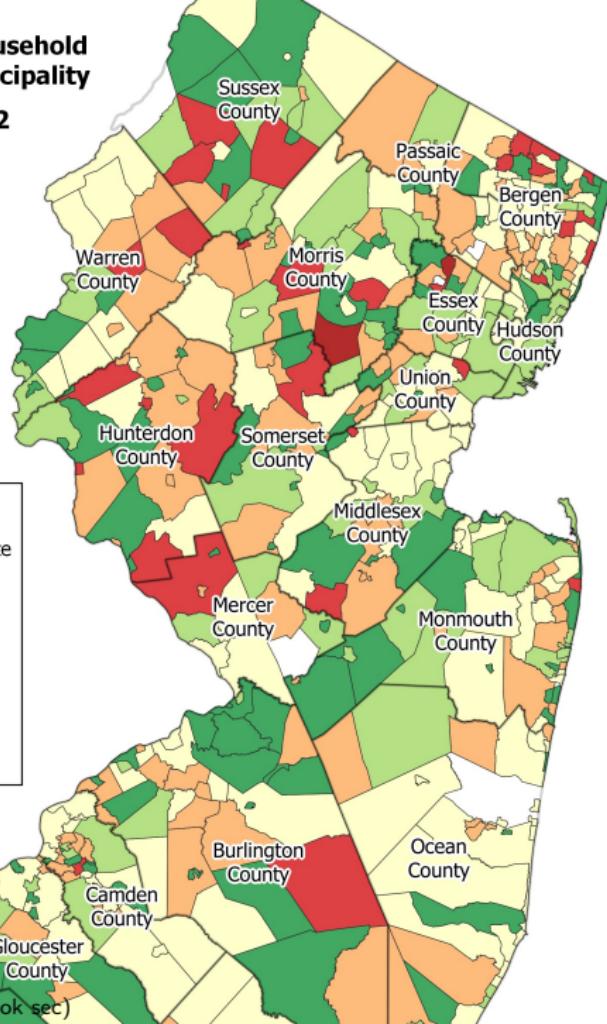
Primary Diagnosis			
Rank	ED	Inpatient	
1	abdominal pain (789.0)		live birth (V33.0)
2	acute URI NOS (465.9)		chest pain (786.5)
3	chest pain (786.5)	congestive heart failure NOS (428.0)	

exemplary examples (see also notebook sec)



Percent Change in Household Poverty Rates by Municipality

From 1999-2012



Legend

Percent Change In Poverty Rate

- 100% - -9.1%
- 9.0% - 9.1%
- 9.2% - 46.2%
- 46.3% - 137.6%
- 137.6% - 499.9%
- Greater than 500%
- No data available

outline

theory (Tufte)

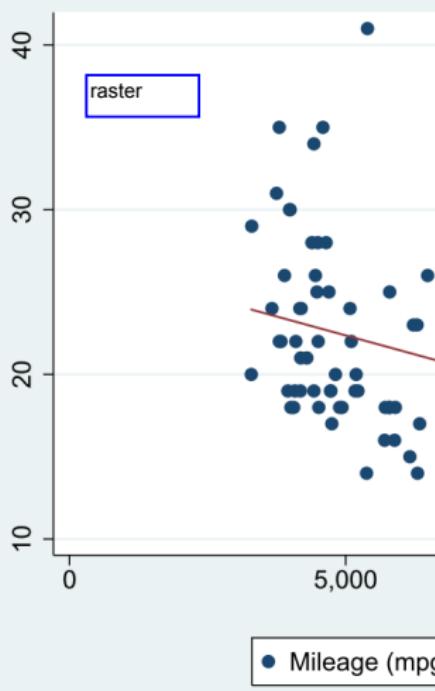
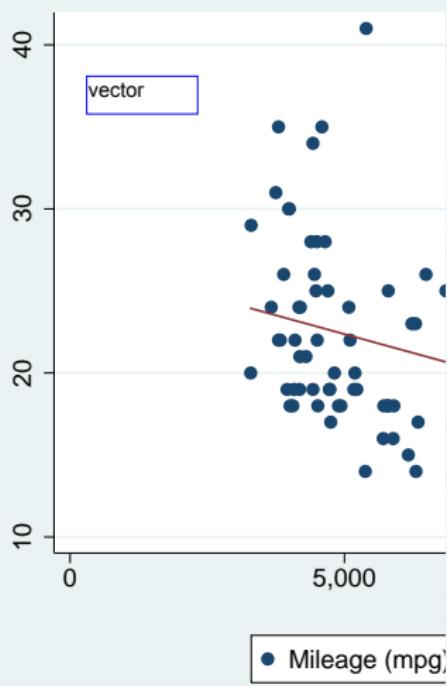
exemplary examples (see also notebook sec)

graph basics

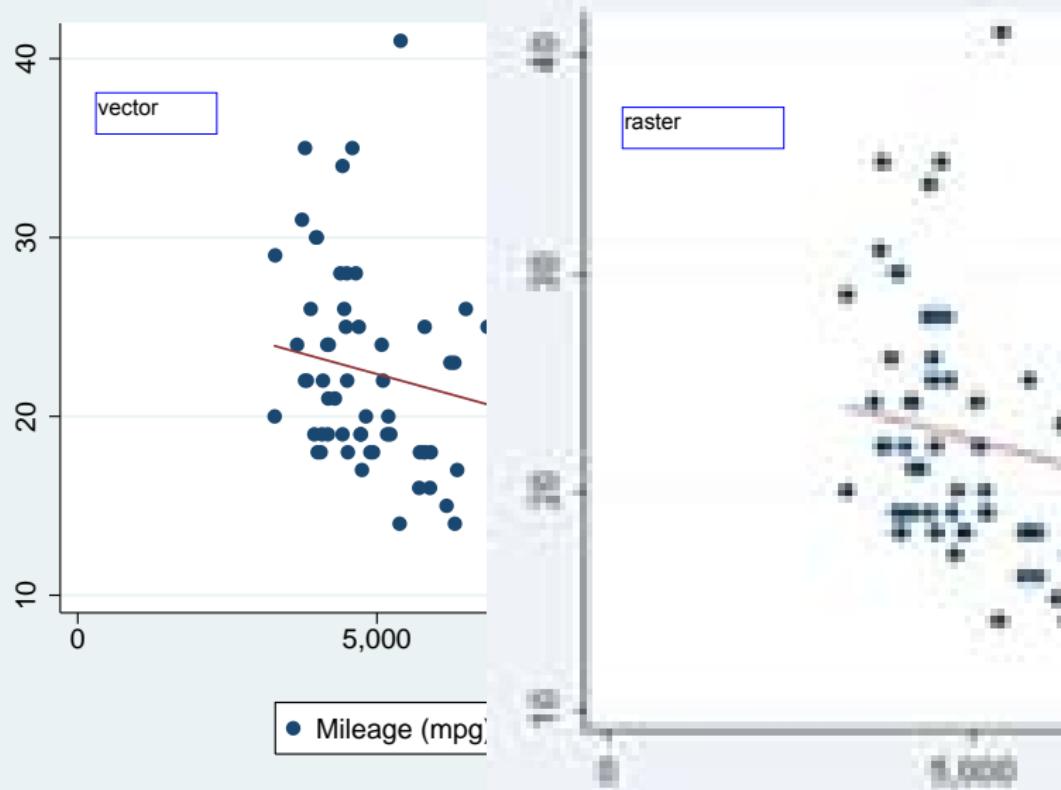
actual graphing

general programming rules, esp scientific computing
elaboration, details, in other words
and yet one more set of rules
the zen of Python

vector vs raster , 100% zoom



vector vs raster, 500% zoom



some graph formats (Py has more!)

- raster: PNG, TIF (specify dpi)
- vector: **PS, EPS, SVG, PDF**

strategy

- sometimes complicated code esp for fine-tuning/customization
- and usually many graph options, impossible to memorize
- way easier to figure it out with
 - examples/galleries (syllabus sec galleries)
 - or examples i gave you in ipynb
- also in colab hover over function it will popup syntax/options
- google it!

make code readable

- use comments
- split one function over multiple lines for readability
- yes do copy-paste from me, internet, chatGPT, etc
 - but then adjust it, clean it up, streamline, simplify

outline

theory (Tufte)

exemplary examples (see also notebook sec)

graph basics

actual graphing

general programming rules, esp scientific computing
elaboration, details, in other words
and yet one more set of rules
the zen of Python

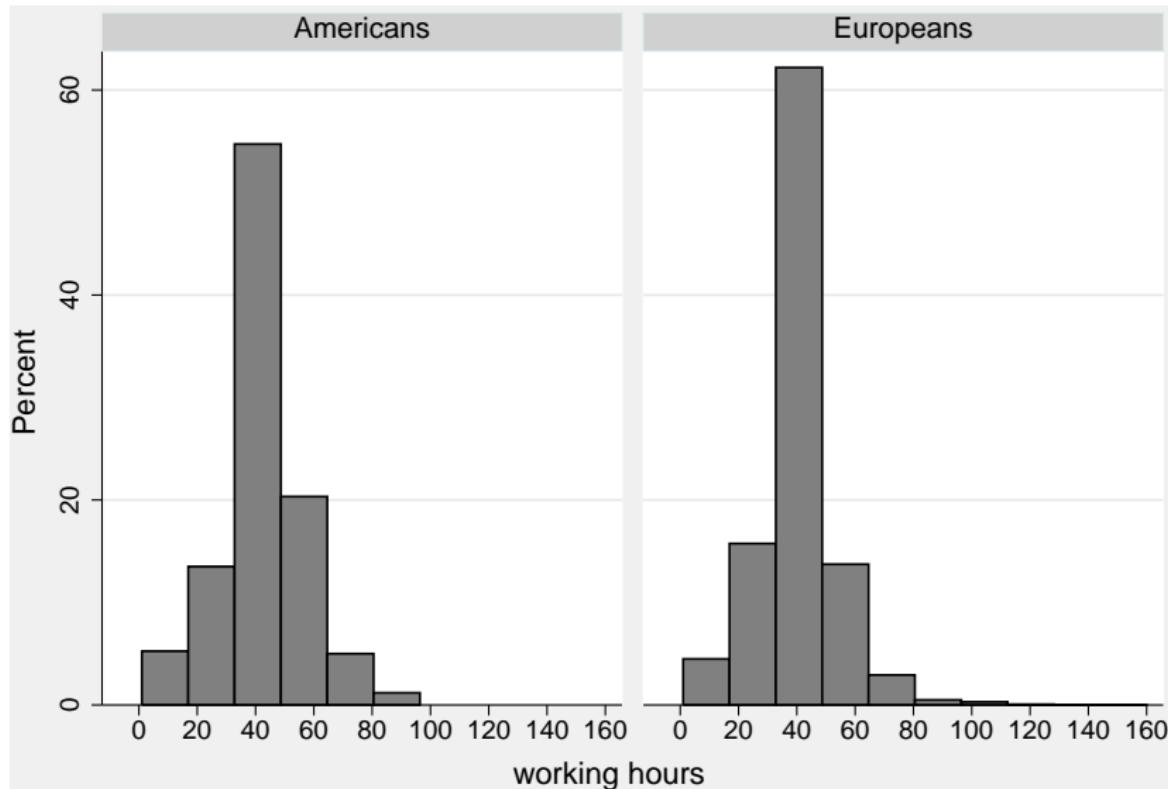
howto graph it?

- again, look at the galleries to get inspired!
- those in syllabus and my ipynb are broken down by cat distribution / hist; ranking / bar charts, etc
- but also consider lev of measurement:
 - continuous
 - distribution
 - categorical, continuous, summary statistics
- lets break the discussion here in this way

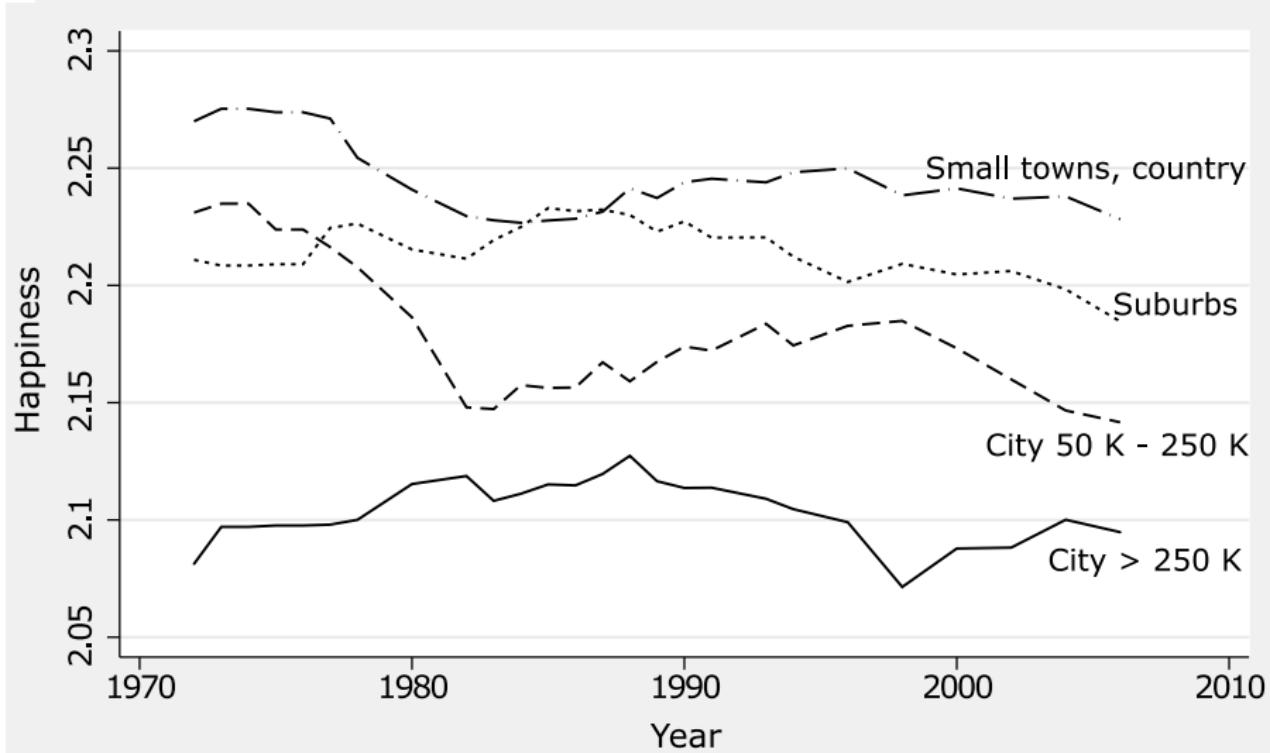
continuous vars

- continuous vars: scatterplots
- time series: lineplots
- combine a bunch of scatter plots using
- show range

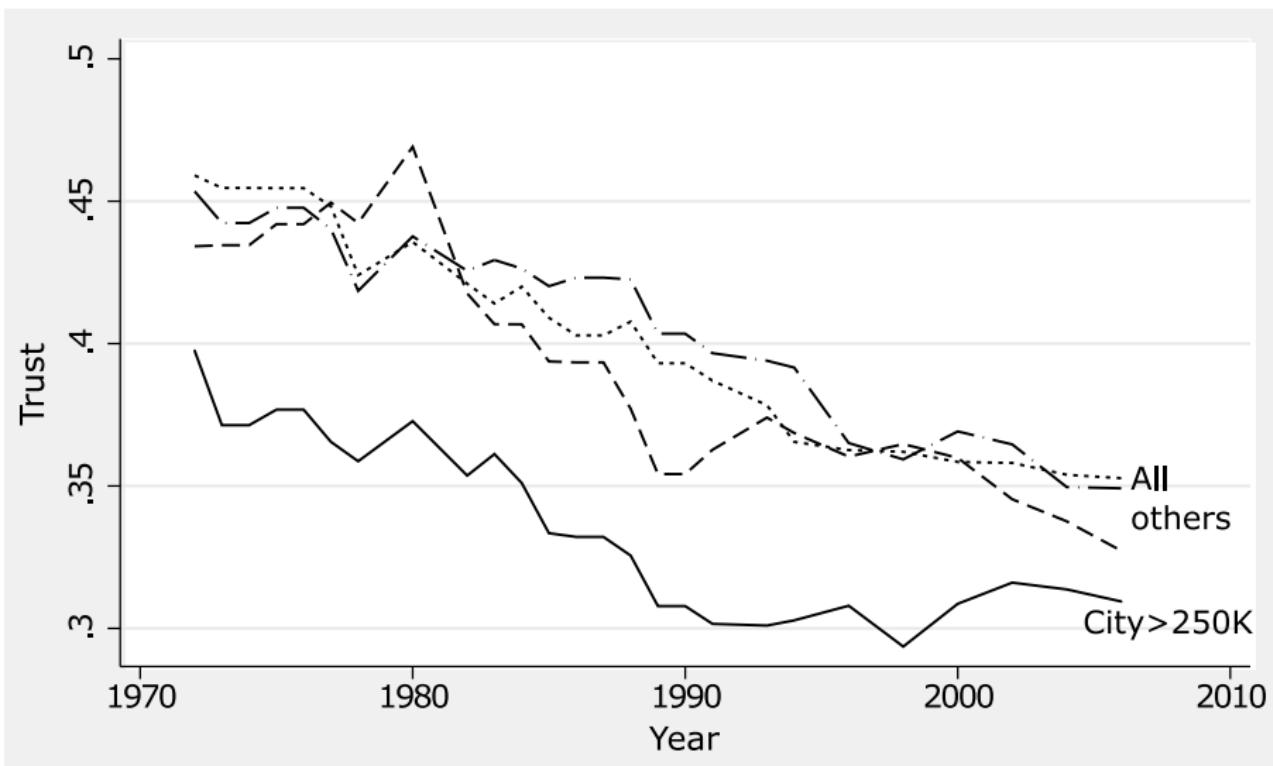
dist ex (subplots)



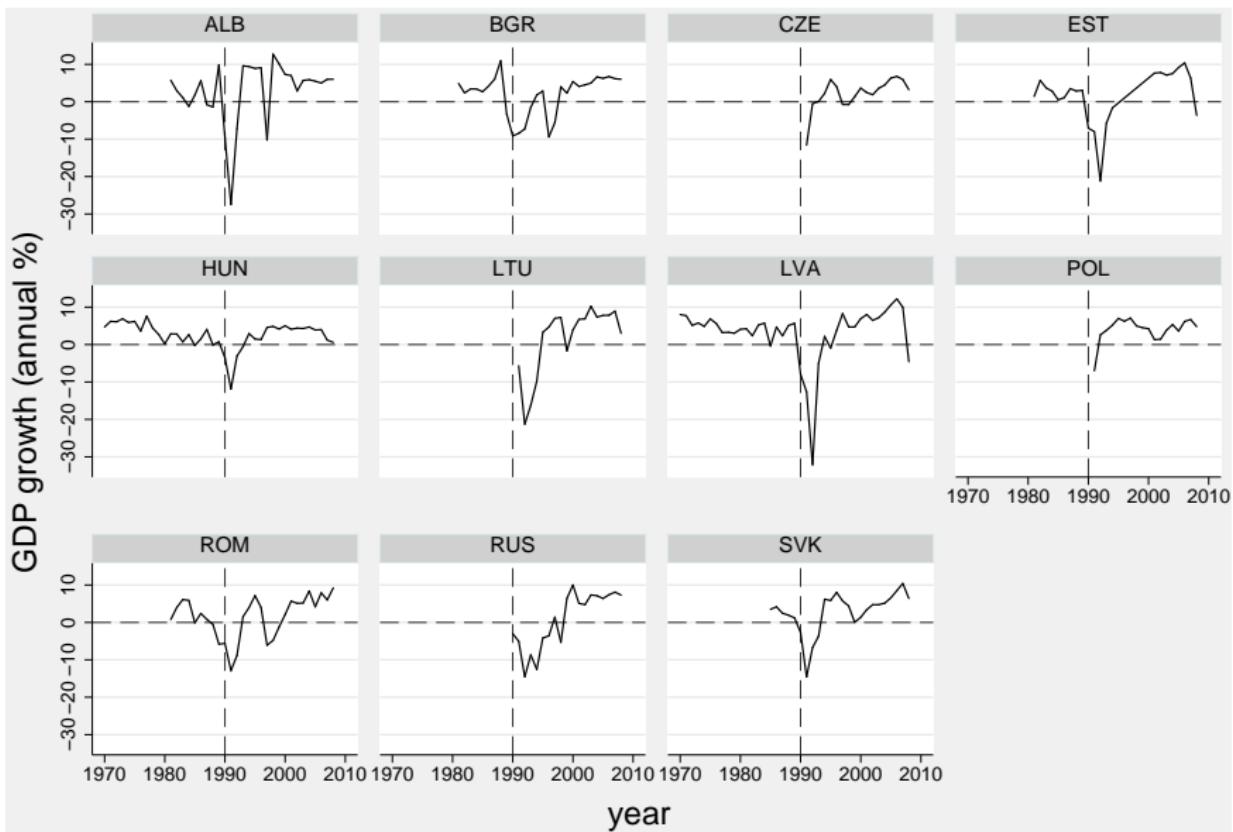
line ex



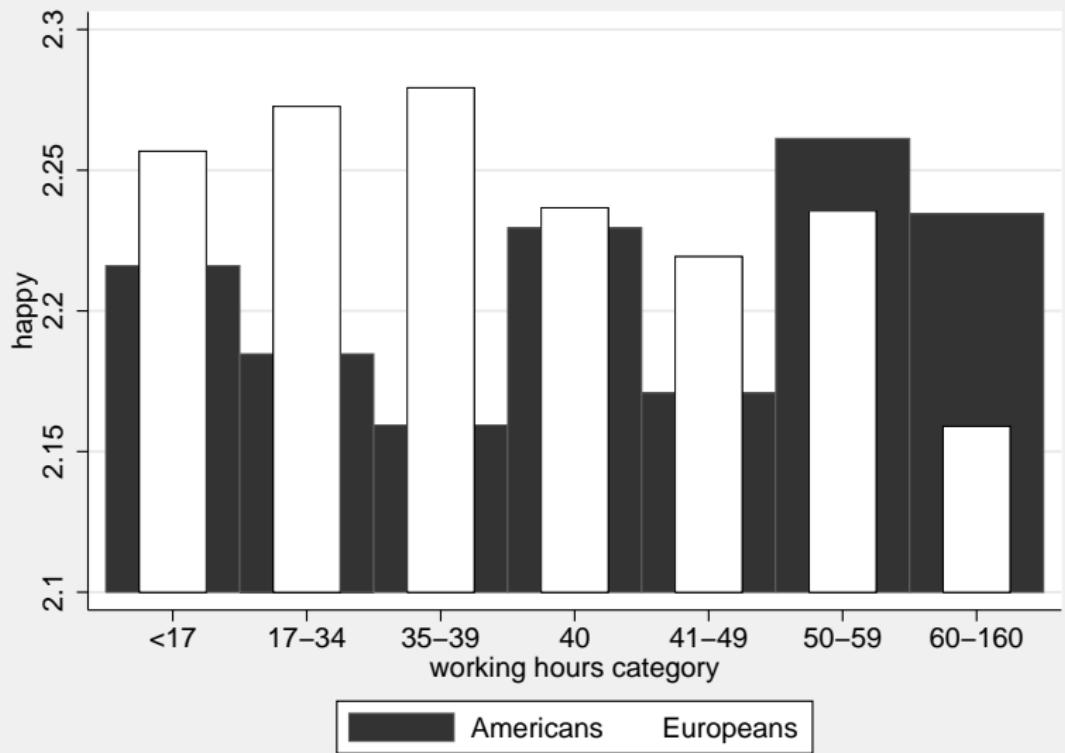
line ex



line ex (mpl: groupby+loop)



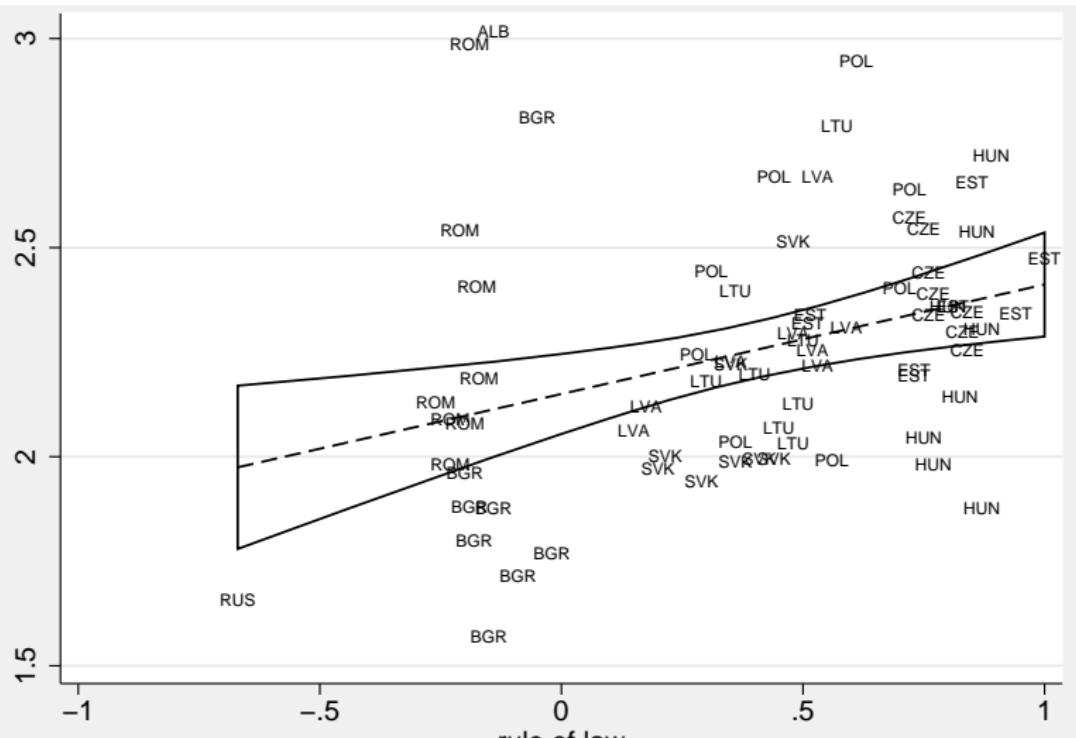
categorical, continuous, sum stat (bar overlay)



postestimation graphs

- why graphing postestimation results
- graphs provide more information than numbers
- graphs show the relationship better than numbers
- again, graphs are especially good for presentations

fit graph example 2



good practices

- use graphs as much as possible, ditch all tables
- BUT avoid graph padding and within-graph data padding
- be as simple as possible (never use chart junk)
- the fewer graphs the better (like nature, science)
 - but to have few awesome ones to share with the world, first have to have dozens in notebook (so clearly mark the main ones v the auxiliary/robustness checks ones) (for class presentation mostly focus on main ones, the story)
 - otherwise your vis won't be robust/bullet-proof/thought-through
- display measures of uncertainty, typically 95%CI

outline

theory (Tufte)

exemplary examples (see also notebook sec)

graph basics

actual graphing

general programming rules, esp scientific computing
elaboration, details, in other words
and yet one more set of rules
the zen of Python

Wilson put it extremely 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 variations on these, and other general rules follow

Know Your Data!

- simply can't use it well if you don't know it well
 - (not just data; the field: theory, lit, etc)
 - this is where you beat IT folks (MS/PhD just in programming)
- 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)
 - or ideally both!
- and this is where vis comes in—best way to get to know your data!
- but 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

simplicity and cleanliness!

- keep it as simple as possible
 - especially if overwhelmed or struggling
- say retain only 5vars and 25obs
 - much easier to understand such data
- simplicity transparency clarity:
 - use fancy code: macros, loops and ados iff they simplify
- have chunks of code only once
 - use root .do, macros, loops, ados to accomplish that
- code it all from raw to final (replication principle)

be fast/efficient and organize

- 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 dofiles)
 - that can be reused for different projects
- don't reinvent the wheel—google often

accuracy or correctness

- ◊ it's fundamental and obvious: vis cannot be 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

efficiency: few lines of code do many things

- ◊ think how you can optimize your code
- 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
- instead of mpl: seaborn, pandas, 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

rewrite/revise

- do “free writing” with code, too (i often come up with some idea out of sudden, and then just write it down...)
- start simple and keep on adding things
- rewrite/revise your code
- improve, add, modify, optimize
 - (there is often a tendency to over optimize, ie spending days on small chunk of code that does not really matter that much)

simplicity: different, often opposite, from efficiency

- ◊ people don't realize this!
- be as simple as possible in writing the code (papers, too)
 - ◊ the more code you have and the more complicated it is:
 - the more likely you have mistakes
 - and the more difficult it is to find them
 - do not complicate your code for the sake of fanciness
 - yes, we do it all the time! don't do it! simpler is better
 - eg I hate my kludge with scatterplot labels, mistake prone, but looks like nobody wrote a function/wrapper to have it easy

standardize

- ◊ standardize— >fewer mistakes (eg make fewer decisions, like a template, on autopilot)
- standardize— >code more transparent, 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
- 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

document

- have text fields in ipynb and # comments in code fields
- meaningful commit messages in git
- 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

portability

- import libs at the beginning, the fewer the better
 - i overdid, i need to cut down
- you could get version of key libs, ie pandas and mpl and save info
- but just 2 libs, can easily trace it back if sth doesnt run in couple years
- but do always save raw data! likely in couple years dataset may disappear or change
- also keep in mind magics (%matplotlib) for running in different environments eg spyder

tradeoffs: life is difficult

- ◊ simplicity is sometimes inversely related to efficiency
- say in programming (loops)
- ◊ simplicity is usually inversely related to automation
- ◊ so make choices
- the more serious you are about coding
- the more you should care for automation and efficiency
- the more work you do,
- the more automation/efficiency 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]

more principles

- from 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
 - 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 and macros, consider ado file

modularity

- “write simple parts that are cleanly connected”
- “controlling complexity is the essence of computer programming”
 - debugging dominates development
- eg:
 - better many small loops that each do one thing than one huge (>100 lines) loop that does everything
 - clear sections of one dofile
 - or many dofiles instead of one dofile without sections

modularity

- code should be organized logically not chronologically
 - do free writing, but then reorganize
 - like with papers, code should be rewritten, eg:
 - no data management in data analysis part
 - move "generate, recode" to the beginning

composition

- “design programs to be connected to other programs”
- notebook or its sec will produce output for another notebook or sec
- eg: you clean up data in one dofile to make data ready for another one to vis
 - or just have one big file
- but 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
 - 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

save time: reuse, don't reinvent the wheel

- if someone has already solved a problem once, reuse it!
- it is 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 you need to do is to adjust somebody else's code or your old code

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 programming

- “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
- that's also why its so important to interpret critically your vis, if something looks funny or unlikely, maybe there's a mistake

- 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!