RAK-19006 Various Topics of Civil Engineering: **Python 3 for scientific computing**
Spring 2018, TUT, Tampere

Published 7.5.2018

Exercises, *lectures 5–6, 9–11*

(The final set. No parallel computing (lec. 7) or HPC (lec. 8) – keeping installation requirements light.)

As a refresher for questions 1–4, see lectures 5 and 6, and this tutorial by Justin Johnson.

For Matplotlib specifically, recall that the module is *matplotlib.pyplot* (which is customarily imported as *plt*); then use *help()*, search the internet, and refer to the gallery.

1. ★☆☆ *NumPy and SciPy.*

   a) Compute x^2 for $0 \le x \le 1$. Use array operations (no explicit looping!). (*Hint*: *np.linspace*)

   b) Create a random square matrix $\boldsymbol{A}$ (size n by n), a random vector $\boldsymbol{b}$ (length n), and solve the linear equation system $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$ for $\boldsymbol{x}$ using NumPy. (*Hint*: *np.linalg*, *np.random*.)

   c) You have a linear equation system that happens to have integer coefficients. How do you force the datatype when you create your matrix $\boldsymbol{A}$ as a NumPy array, so that NumPy will treat the numbers as floats?

   d) Look at the online documentation for scipy.linalg. Roughly, what is the main difference between *scipy.linalg* and *np.linalg*?

   e) How do you solve an eigenvalue problem $\boldsymbol{A}\,\boldsymbol{x} = \lambda\,\boldsymbol{x}$ with SciPy? If your matrix $\boldsymbol{A}$ is general? If hermitian? How do you solve a generalized eigenvalue problem $\boldsymbol{A}\,\boldsymbol{x} = \lambda\,\boldsymbol{B}\,\boldsymbol{x}$ with SciPy?

   f) How do you perform an SVD (singular value decomposition) in NumPy and/or SciPy? How do you get the condition number of a matrix?

2. ★☆☆ *Basic plotting*.

   a) Using NumPy and Matplotlib, plot the monomials x, x^2, x^3, x^4 and x^5 into the same picture, with x in the interval [0, 1]. (*Hint*: in Matplotlib, *hold* is enabled by default.)

   b) Add a grid, x-axis and y-axis labels, and a title.

   c) Label each curve (recall that you can use TeX math in the labels) and add a legend.

   Don't provide labels directly to the *legend()* command. How to make sure, in your source code, that each label definitely stays together with the curve it is supposed to label?

   (*Hints*: raw strings are convenient for TeX code, e.g. r"$x**2$"; see the kwargs of *plt.plot*.)

d) Save the resulting figure in a format of your choice (e.g. *pdf*, *svg* or *png*).

Do this from your script, so that the saving occurs automatically every time the script runs, without the need to press any buttons.

(*Hint*: *plt.savefig*.)

> ※ The *eps* exporter is broken-ish, not recommended. If you intend to use the figure in a document rendered with pdflatex, just export as *pdf*; pdflatex can use *pdf* figures.
>
> If you intend to manually add annotations (but see the gallery for how to script things like that!) in Inkscape, save as *svg*.
>
> Save as *png* only for online use; bitmaps, unlike vector graphics, don't scale well!

e) [optional] In your opinion, how many samples is enough for plotting a curve? Why?

3. ★★☆ *Intermediate plotting*.

a) *Controlling the look*. Look at the *linestyle*, *linewidth*, *color*, *marker*, *markersize* options of *plt.plot*. Try them out on your solution to exercise 1.

b) *Subplots*. Look at *plt.subplot* (not to be confused with *plt.subplot**s***.) Make a 2-column by 3-row array of subplots, and plot a different function in each. Add a title for the whole plot (see *plt.suptitle*). Try adding axis labels for each subplot.

c) Placement of *y-axis labels*. In a two-column subplot like in **b)**, the *y*-axis labels often don't fit. Place the *y*-axis labels of the right column of subplots onto the right side.

(*Hint*: https://stackoverflow.com/questions/13369888/matplotlib-y-axis-label-on-right-side

You can get the axis object you'll need here as the return value when you call *subplot*.)

d) *Independent y-axes*. In the interval [0, 1], plot the functions

$f(x) = -6\,x^5 + 15\,x^4 - 10\,x^3 + 1$
$g(x) = (1 / 2) \cdot (-x^5 + 3\,x^4 - 3\,x^3 + x^2)$

into the same picture, *using an independent y axis for each*. The vertical scale for *f* should be placed on the left, and the scale for *g* on the right.

(*Hint*: *twinx* method of axis objects; this is completely unrelated to item **c)**.)

> ※ These are Hermite polynomials from lecture 5, slide 20; code available on GitHub.

4. ★★☆ *3D plotting*.

   The 3D plotting API of Matplotlib is a bit clunky; feel free to use the example from lecture 5 (or alternatively, anything suitable you can find on the internet) as a template.

   a) Using NumPy and Matplotlib, plot the *wireframe* of $f(x, y) = \sin(\pi x) \cos(\pi y)$ for x and y in the unit square.

   (*Hint*: use NumPy's *meshgrid*, similarly to how this would be done in MATLAB.)

   b) Add axis labels and a title. Set a nice camera angle.

   c) Experiment with the options of the wireframe command to get a nice-looking plot.

   (*Hint*: *strides*. You can use a denser mesh for the data than the wireframe grid will be; the extra data will be used to make curved edges for the rectangles, to make the lines follow the actual shape of the surface more closely than if just linearly interpolating the rectangle vertices.)

   d) Plot the same function as a filled surface (instead of a wireframe).

   e) As an alternative view to the same data, make a 2D plot (seen from the top) as a colorsheet. Add a colorbar.

   (*Hint*: *pcolor*, *pcolormesh*. Compare *contour*.)

5. ★★☆ *Version control*. (Lecture 9)

   In the lecture notes, pp. 80–88, there is a step-by-step crash course on the basics of the *git* DVCS (distributed version control system).

   > ※ The “D” simply means that at all times, you'll have a complete copy of the project history *also on your own computer*, not just on the version control server.

   a) If you haven't used git (or especially, if you haven't used any version control system), walk through the crash course and try out *git* in practice.

   (As test data, you can use any plain text files – doesn't have to be code unless you prefer.)

   b) [optional] If you haven't used GitHub, read the Hello World guide and try out the basics.

   > ※ Item **b)** requires creating a free account on GitHub; only makes sense to bother with it, if you think GitHub could be useful for you.
   >
   > In a nutshell, GitHub is a “social medium of version control”; a collaboration site for open-source projects. But you can also use it to just host your own projects (regardless of whether you intend to collaborate with anyone on them), provided that you intend the source code to be public. (They also offer private repositories, but that requires a paid membership.)

6. ★★★ ***Let** it be*. (Lectures 10–11)

   In the solutions to exercises 1–2, question 4 e), we noted that Python lacks a **let** construct.

   Actually, that can be fixed, sort of. Consider this code:

   https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/let.py

   Analyze how it works. Specifically:

   a) What is the role of each nested **def** in *let_over_def*? How do the bindings – which are provided at the definition time of the function being decorated – travel to the actual call(s) to the function, which occurs later?

   b) *On the composition of decorators*. How is the combined decorator **let** built out of the components *let_over_def* and *immediate*?

   (*Hint*: keep in mind how a decorator desugars in Python; from that, everything follows. The point is to think in terms of how things work behind the scenes.)

   c) How does **letexpr** work?

   Finally, a small implementation trick: how does it single out the *body* parameter, out of the arbitrary arguments it accepts?

   Then, on a more general note:

   d) If we stick with named **def** blocks used in the function-definition-as-a-code-block spirit, **let** is almost redundant. Explain why.

   (*Hint*: look at **letify**. If our only aim is to write let-ish bindings in a readable manner, do we really need even that?)

   e) [optional] Out of the given implementations, do you prefer **let** or **letify**? Why?

   f) [optional] In your opinion, are these constructs useful in Python, or not? Why?

   (There are no “right” answers to items **e)** and **f)**; this is software engineering.)

   g) [optional] In Lisps, **let** is implemented as a macro that desugars into a λ that is then immediately called. What, if anything, do decorator factories and macros have in common?

7. ★★★(★+) *Monads.* ***In Python.***

   A monad is just a monoid in the category of endofunctors, what's the problem?

   Seriously though, as our final exercise topic, let's look at the **monad**, a concept from the pure FP world – particularly Haskell – without leaving Python. (How many ★ this question deserves depends on how deep you want to go.)

   This topic was not covered in the lectures. In lecture 10, slide 4, we just mentioned in passing that a thing called a "monad" exists, and that in Haskell it's used to encapsulate side effects.

   However, that's not the full story, and not even the main feature that makes monads interesting from the viewpoint of software development.

   Let's concentrate on the basics, and look at monads as **containers**. *What do multivalued (mathematical) functions, error handling, and debug logging have in common?*

   A design pattern lurks, waiting to be abstracted. Take a look at one or more of these materials:

   Stephan Boyer (2012): Monads, part 1: a design pattern
   Nikolay Grozev (2013): Monads in 15 minutes
   Stephan Boyer (2014): Super quick intro to monads
   Dan Piponi (2006): You Could Have Invented Monads! (And Maybe You Already Have.)
   What is a monad? (on StackOverflow)

   And optionally, also:

   Stephan Boyer (2012): Monads, part 2: impure computations

   Now for the questions!

   a) Very roughly, and very briefly, *from a software engineering viewpoint* (instead of a mathematical, *category theory* viewpoint), **what is a monad?**

   b) Why should you make sure your own monads (if you define some) satisfy the monad laws?

   c1) [alternative to c2)]

   There's an OO(F)P-ish implementation of the Maybe, List, Writer, State and Reader monads at

   https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/monads.py

   Analyze how (at least some of) it works, to learn some basics about monads via examples.

   (*Hint*: it may be useful to keep in mind – in a rough manner – the expected type of each object. Especially so since Python is dynamically typed, so it doesn't help you in this department, unlike Haskell. The idea being primarily based in a language with static typing, it's not surprising that it is most convenient to use in a language with static typing.

For example, ***bind*** needs an *x* (in the code example, stored in *self.x*), of type *M a*, and an *f*, of type (*a -> M b*). The output of *bind* is of type *M b*. Here *a* and *b* are any datatypes admissible for whatever is being computed, and *M* is the type of the monad. This kind of analysis can help understand what the operations do: essentially, *bind* unpacks the data from *M*, runs it through *f*, and then re-packs the result into a fresh instance of *M*.

On the other hand, the above is a very abstract way of thinking about a very common operation. As the simplest examples, consider the Maybe and List monads. What is actually being done when they *bind*? What would you have to do manually, if you did the same without monads?)

c2) [alternative to c1)]

Implement some monads on your own, in Python – perhaps based on one of the above explanations – to learn some basics about monads in practice.

(*Hint*: *Maybe*, *List* and *Writer* (in that order) are perhaps the easiest place to start – and List already gives a very handy tool for nondeterministic evaluation, also in Python. *State* and *Reader* require a different viewpoint.)

Finally, if you want to get serious about this stuff in Python, here are some links:

Functors, Applicatives, And Monads In Pictures - in Python
Ø: Functors, Applicatives, And Monads in Python
Pynads - "Dumb implementation of monads in Python".
PyMonad
Dan Piponi (2008): Monads in Python (with nice syntax!)
Monad-Python

And if you are curious about this idea in its natural habitat, here are some more links for that.

First, be sure you understand how to read the ***do notation*** [1] [2] – a simple sugaring that makes monadic pure FP code look like imperative code.

Monads as containers
Monads as computation
Monads with Join() instead of Bind() (on StackOverflow)
Bartosz Milewski (2016): Monads and Effects

Learn You A Haskell: Functors, Applicative Functors and Monoids
Learn You A Haskell: A Fistful of Monads

What a Monad is not
Do notation considered harmful

Yann Esposito (2012): Learn Haskell Fast and Hard
Haskell/Category theory